Bright – A C-like Lua Derivative

Terry Moore
tmm@mcci.com
MCCI Corporation
Who we are

- System engineering company, specialists in USB technology
- Ninety people
- Headquartered in Ithaca, NY; sites in Austin, Tokyo, Taipei, Seoul and Europe
- Focused on cell phone industry
  - Over 500 million cell phones that use MCCI technology
  - Two of the top four cell phone OEMs
  - Two of the top four cell phone Platform Vendors
- Additional markets in set-top boxes, car navi systems
Focus of Presentation

• What did we learn about Lua based on the changes we made?
• How are we using our re-skinned Lua
MCCI’s Problem Space

• Our customers are huge engineering teams
  – many products, shipped in high volume
  – years of prep work for one month’s production!
  – very risk averse

• Our software has to be integrated into their development environments
  – each environment is different
  – each environment evolves unpredictably and asynchronously

• We have to maintain economy of scale and deliver bug fixes across all the different consumers

• We use automation intensively
What Automation to Use?

- Java, Perl, Python, etc., are “well accepted”
  - For brevity, let’s say “LDJ” for “language de jour”
- If they’re not using LDJ, all these LDJs are very heavyweight; this generates resistance to using our automation
- If they ARE using LDJ, they’ll have their own version, and it won’t in general be the same as the version we’re using or the version any other customer is using
- Most LDJs have enormous libraries which add to the complexity
Why not Lua?

• Lua is a research language, targeting embedded scripting
  – needs to evolve
  – backward compatibility is less important than exploring new ways of saying thing
  – Lua as a stand-alone language is secondary to Lua as an embedded language

• MCCI needed a language that would emphasize backwards compatibility and stand-alone tool applications
  – backward compatibility is critical
  – Bright used as a stand-alone language is a primary use-case
  – Lua 3.2 to 4.0 made us realize that in order to use Lua technology, we needed a degree of independence
What did we change and why?

• We liked Lua a lot – we hoped for its general adoption – and we wanted to stay out of the way...
• We changed syntax – something “almost like” Lua seemed worse than something quite different
  – we switched to C-like syntax for somewhat cynical reasons
• We changed semantics to meet the need of a production environment
  – Zero-origin indexing
  – “Undefined” values
  – No locale sensitivity
  – Empahsis on script portability over functionality
• We changed the command-line wrapper programs (bright.exe and brightc.exe) to be more like the Unix equivalent tools
• We changed the externally visible names of all the C API namespace entities so as not to collide with Lua.
• We added some things we liked
Three kinds of changes

• Trivial – nothing interesting about them
• Small – somewhat interesting, but not a major change to the flavor of the language
• Large – major changes to the flavor of the language
• Curiously, the effort involved was inverse to the scale of the change
Trivial Changes

- C-like syntax
  - This was trivial, in the sense that it was a simple exercise in the lexer and parser
  - More details as to what we did later – if there’s time
- Created man pages
- Wrote a reference manual (adapting liberally from the Lua reference manual)
- With C-like syntax we got bit-wise operators – enormously convenient
  - Of course, have to convert to LONG first
Index origin zero is a very small change, conceptually

– You can write Lua or Bright without knowing the origin, if you’re careful

```lua
function GetOrigin()
    for i,v in {1} do
        return i;
    end;
end;

_ORIGIN = GetOrigin();
```

– Then enumeration of an array can be written as, e.g., (in Lua)

```lua
for i= _ORIGIN, #t - _ORIGIN do ....
```

Zero-origin makes strsub() less convenient to use, however, as there’s no pleasant zero-origin mapping unless you use -2 as your start point for negative indexing
Small Changes

• “.〈id〉” notation distinguishes reflexive use of strings from “normal” strings
• Changes to wrapper executables for “stand alone” use
  – Add “-c” option for symmetry with “sh –c”
  – Allow #! prefix in compiled scripts
  – Allow multi-chunk compilation (and teach compiler to produce the #! prefix)
  – Add fallback “main()” invocation in the bright.exe wrapper
Large Changes

• Adding Undefined, and making NULL a valid key and datum for tables
  – code changes were relatively minor, one day’s work
  – flavor of language changed substantially
  – If NULL is a valid key, then NULL cannot be used as the distinguished “end” value when iterating over tables
  – If NULL is valid datum, then presence/absence testing requires extra linguistic features

• The VMs were compatible up to this point (sigh).
What we learned

• Making a more C-like language substantially reduced resistance to adoption in MCCI’s community
• Changing to zero origin reduced errors for programmers switching back and forth from Bright to C
• The “undefined” value makes programs fail early on typos – as desired
  – Works very well for global and local typos
  – Returning “undefined” for missing table entries similarly makes programs more robust
  – Productivity and reliability went up noticably & immediately
• Changing tables to have NULL (nil) as a first-class value is very convenient
  – but it really changes the implementation and style substantially
• Bit-wise operators are EXTREMELY convenient (even if lua_Number is a double)
  – Lua should add these
• Having a C-like syntax allows for some “clever hacks” when checking/using complex #include files
Why name it “Bright”?  

• It’s sort of a pun  
  – Lua in Chinese is 月.  
  – If you add sun to moon, (日 + 月) you get the character 明, ming2, meaning “bright”.  
  – Ming was already taken, hence...
How do we use Bright?

• As a Cross-platform Programming Language
• Rapid Prototyping
• Shell scripting
  – we use it like awk
• Embedded Scripting
• C Header-File Crunching
Cross-platform Programming Language

• documentation generation
• source release generation
• automatic dependency generation for our build system
• The minor changes made to lua and luac were very helpful
Rapid Prototyping

• Problem: remote customer with broken hardware and only a Tektronix scope
• Solution: built a tool to recover USB high-level data from only a differential trace of the data lines
  – differential-to-single-ended conversion
  – phase-lock loop for clock and data recovery
  – NRZI to normal data
  – CRC calculation
  – Token recognition
  – Total effort (since it was built step-by-step): about 4 hours. This would take a week in C.
• For low-level hardware operations, the bitwise operators of Bright are extremely useful
Embedded Scripting

• MCCI’s cross-platform version of NetBSD `make(1)` supports scripting in Bright.
  – extremely convenient because it removes dependency on external computation tools for complex `make` operations
  – allows us to have one `make`file that works anywhere, for any target
• MCCI’s `usbrc` tool compiles USB initialization code from high-level descriptions – we use Bright for scripting information about hardware limitations
• All of MCCI’s USB test applications use Bright as the test scripting language
• MCCI’s version of `usbview` uses Bright to learn how to decode device class descriptions
C Header-File Crunching

• It’s easy to generate a Bright program from a well-formed header file
• This makes it easy to do certain kinds of tests on header files, and to use C definitions in Bright scripts
• We use this, for example, for an assembler for a special purpose kernel VM “mcciport.sys”.
Future Directions

• Complete module system – somewhat different than Lua, as the goal is to eliminate first-order “globals”
• 64-bit integers
• **try** – explicit exception handling
  – using `call()` for this is clumsy
  – nothing as elaborate as C++ is intended
• Optional stronger typing
  – internally implemented version of our CreateClass facility (again, for productivity)
• Steal features from Lua 5.1 (# operator, iterators)
• Make the lexer available directly
Supplemental Slides
What did we change and why?

• We liked Lua a lot – we hoped for its general adoption – and we wanted to stay out of the way...
• We changed syntax – something “almost like” Lua seemed worse than something quite different
  – we switched to C-like syntax for somewhat cynical reasons
• We changed semantics to meet the need of a production environment
  – Zero-origin indexing
  – “Undefined” values
  – No locale sensitivity
• We changed the command-line wrapper programs (bright.exe and brightc.exe) to be more like the Unix equivalent tools
• We changed the externally visible names of all the C API namespace entities so as not to collide with Lua.
• We added some things we liked
Changes to Wrapper Executables

• Lua 4’s wrappers were too simplistic for production use
  – Most important: changed brightc (luac) to combine multiple input files into a single output file
    • compiled script elaborates byte code for each file in turn
  – Changed bright.exe (lua.exe) to invoke global function main(ARGV)
    • only if the global chunk doesn’t return an explicit value
    • only if main() is defined
  – Allowed #! as first line of compiled (.bro) scripts
  – Minor changes to command line options
What Lua things are missing?

• New features added in 5.0 and 5.1
  – Up-values are not general, and use the Lua V4 syntax
  – No threads
  – Nestable long-string constants
  – Boolean value support was added “differently”; no boolean type
  – The # operator (good idea, that)
  – The new module support
  – Weak tables
  – Library improvements

• Automatic conversion between strings and numbers

• Locale sensitivity for program text
  – a program has the same meaning, no matter the locale in effect at parse time
New semantics

• A new type was added: undefined, with a single distinguished value, (also called “undefined”). All variables initially have value undefined.
  – Any attempt to evaluate an undefined value results in an error.
• Table semantics are extended
  – nil (bright: NULL) is a valid table index, and a valid table value
  – If an index value is not in an array, the result is the undefined value
  – New expression syntax: <v1> in <v2> allows an easy way to check whether <v1> is a key in the table expression <v2>
  – Entries must be removed using tdelete(t, k) -- t[k] = NULL no longer removes index k.
What C things did we add?

• Language
  – All binary and ternary functions from C:
    • bitwise &, |, ^, <<, >> -- we force numbers to integer, do the bitwise math, then return to float format.
    • ISO e ? v1 : v2 and gcc e ?: v
  – The <iso646.h> alternate tokens
  – The alternate token spellings from ISO C (writing “<%” for “{”), and so forth.
  – for(;;) {} and do {} while ()

• Extras
  – TRUE, FALSE, NULL are reserved words, and predefined.
  – All the reserved words from C++ are also reserved words in Bright
What C syntax did we change?

• Comma is used for multiple assignment, not multiple expression evaluation
  – \( x, y = f(), g() \) is three expressions in C: evaluate \( x \); assign \( f() \) to \( y \), and evaluate \( g() \).
  – \( x, y = f(), g() \) is two expressions in Lua and in Bright: evaluate \( f() \), evaluate \( g() \), then assign respective results to \( x \) and \( y \).

• Exponentiation is useful; we kept it (but use “**” instead of Lua “^”).

• Concatenation is expressed using “..” rather than more C-like juxtaposition. (But the tokenizer will concatenate literal strings if they’re written side-by-side.)

• Double and single quotes both delimit strings – ‘a’ is the same as "a", not 0x41.

• Functions are defined as in Lua or Awk: \texttt{function f() \{ \} }

• No compile-time types
Dot notation

• The “.<id>” syntax generates the string “<id>”, but expresses the intention that the programmer is providing the name of a key in a table

\[
v = (.n \text{ in ThisTable}) ? \text{ThisTable.n : 0};
\]

– I think I stole this from atom notation in an older Lisp?

• Perhaps a better example:

\[
\text{if (! (.Lib in globals()))}
\]
\[
\text{Lib = dofile("mcci-v1.bro");}
\]

• Makes reflexivity somewhat more explicit – by convention, if you write .foo, you mean foo as an identifier in some kind of reflexive context, whereas "foo" is a string for some kind of external comparison

– can slightly simplify the problem of renaming table indices, if used consistently: a search for “.foo” will find more correct instances than a search for “foo”.

7/15/2008
Built-in Library Additions

• Because of the global namespace issue, we decided to prefix all bright-additions with "bright_".
  – `bright_diropen()`, `bright_dirread()`, `bright_dirclose()` – equivalent to the familiar Unix routines
  – `bright_stat()`, `bright_stat_decodemode()` – portable version of `stat()`
  – `bright_shortpathname()` – returns the [system-dependent] short version of a pathname
  – `date()` was extended in a similar way to some of
What Lua 5 work did we duplicate?

• We added separate environment tables for each function (but did it differently, and more conservatively, i.e. based on the Lua 4 mechanisms)
  – this was done in anticipation of Bright modules, which so far have not been fully implemented

• Miscellaneous: \texttt{break}, hex constants, modulo (\%\%, defined exactly as in Lua 5, and probably for the same reasons)
Bright standard library

• In addition to the normal built-in libraries, MCCI has a standard library of Bright facilities, written in Bright.
• Normally (but not necessarily) referenced as contents of table Lib
• Interesting work
  – Lib.Disclose(), is akin to \texttt{unpack()} from Lua 5.1 – named by analogy with APL.
  – Lib.GetFlags() is a standard command line parsing package
  – Lib.Basename(), Lib.Dirname() are OS-independent filename parsers
  – Lib/CreateClass() creates abstract classes with stronger type checking
  – Lib.CreateStructureClass() creates abstract classes with specific binary representations (for interoperating with other system components)
  – Lib.VectorToString() is like table.concat() from Lua 5.1
Example Lib.CreateClass

cID = Lib.CreateClass(
    .ID,
    {
        { .string,      .sName },
        { .number,      .Id },
    });

cTARFILE = Lib.CreateClass(
    .TARFILE,
    {
        { .generic,     .File },
        { .generic,
          .CurrentEntry },
    });

cTARENTRY = Lib.CreateClass(
    .TARENTRY,
    {
        { .generic,      .Parent },
        { .number,      .HeaderPos },
        { .string,      .sPathName },
        { .string,      .name },
        { .number,      .mode },
        { .number,      .size },
        { .number,      .mtime },
        { .ID,          .uid },
        { .ID,          .gid },
        { .number,      .type },
        { .string,      .linkname },
        { .string,      .prefix }
    });
Design Decisions that Worked

• Adding a default call to main() in the bright.exe wrapper makes large programs look much nicer to C programmers.
• Adding Undefined greatly simplifies debugging.
• NULL as a table value; TRUE and FALSE as synonyms for 1 and NULL.
• We allowed local declarations in for (; ;), much as in ISO C99, which was very nice:
  
  ```c
  for (local i = 0; i < Max; i=i+1) { f(i); }  
  ```

  is more readable (to our C programmers) than
  ```c
  for i=0,Max-1 do { f(i); }  
  ```

  Both, of course, are permitted. (The latter is somewhat faster.)
Drawbacks (what we missed)

• The library routine names should have been mapped more closely onto their C equivalents.
• We should have done more work on modularity, or back-ported the Lua 5 work.
• Our programmers miss compound assignment (+=, etc) and switch()
• strsub()'s semantics are not well adapted for zero origin.
• It would have been nice to have the Bool type
Thanks

• Chris Yokum of MCCI did a lot of library work, and was our first enthusiastic internal user
• The Lua project has been incredibly understanding about our somewhat heretical approach