A bytecode assembler for Tcl

Ozgur Dogan Ugurlu[*]
University of San Francisco, San Francisco, CA
dogeen@gmail.com
Kevin B. Kenny
GE Global Research Center, Niskayuna, NY
kenny@ge.com

Abstract. Tcl, like most dynamic languages, operates by compiling its programs to an intermediate language ("bytecode") that is executed by an interpreter. This paper presents an assembler for the Tcl bytecode language. Prior to the development of the assembler, a Tcl program could generate bytecode only from Tcl scripts (or from C code using unpublished interfaces in the Tcl core). With the advent of the assembler, a Tcl script can generate specified bytecode directly. This ability should enable readier experimentation with compile-time optimizations, with targeting other languages onto the Tcl bytecode engine, and with trading off Tcl's dynamic nature for speed of execution. Some of the challenges faced in developing the assembler are presented, and some preliminary observations about the performance that can be gained from assembly code are made.

1 Introduction

Tcl, like most dynamic computer languages, interprets its programs in two phases. In the first phase, the code is compiled to an intermediate language ("bytecode"), and in the second, a virtual machine interprets the bytecode produce the program's results. The translation of Tcl commands to bytecode happens late in the proceeding – usually at the first time that a procedure is called, or even later – and the bytecode compiler is extremely conservative. For instance, it is careful to preserve source code for recompilation in the event that a core command like `[set]` or `[for]` is redefined while the code is executing. Because of this conservatism, the bytecode is often not as efficient as it might be. Moreover, since evaluating Tcl scripts is at present the only means of generating bytecode, it is difficult for people to experiment with different compilation strategies or with targeting other languages into Tcl's bytecode interpreter.

This paper presents an assembler that will aceept an abstract representation of bytecodes in a Tcl-like syntax and prepare them for evaluation in Tcl. This assembler is the first time that bytecode has been exposed at a user level (even in C!) and is expected to open the door to experimentation with the engine.

Section 2 presents a brief overview of Tcl's bytecode language. Section 3 discusses the internal programming interfaces that support the compilation of scripts and expressions, and how they are adapted to processing assembly language. Section 4 presents the assembly language notation in brief. Section 5 discusses some of the problems that the assembler must solve in order to generate code that will execute safely in the execution engine. Section 6 goes into more detail about potential uses of a bytecode assembler, including some estimation of the performance difference between handwritten assembly code and Tcl. Section 7 discusses the work that remains to be done, and Section 8 presents some conclusions.

* Research sponsored by the Google Summer of Code, 2010

2 How Tcl bytecode works.

The Tcl bytecode interpreter is an abstract stack machine, akin to Forth or PostScript. All of its instructions work by moving operands to and from an operand stack. The commonest instructions move constants and variables to and from the stack, or remove operands from the stack and replace them with results. For example, the Tcl code:

```
set x [expr {$x * 2}]
```

might compile to bytecode like:

```
load x; # move $x to top of stack
push 2; # push "2" on top of stack
mult;   # multiply top two stack
        # elements. Replace with
        # product.
store x;# put top of stack in $x
pop;    # discard top of stack
```

As the just-in-time compiler is processing a script to produce bytecodes, it tracks what distinct constants are present in the script, and places them in an array of Tcl_Obj pointers called the "literal pool." (There is one instance of the literal pool per bytecode compilation; in practice, this translates very nearly to one instance per procedure.) The `push` instruction, in the actual bytecode, contains the index in the current compilation's literal pool of the constant to be stored on the stack.

In a similar way, the compiler maintains a table of local variables known in a procedure. A variable whose name is known at compile time can be addressed by index into the local variable table: the `load` instruction in the bytecode above gives the variable's index in the local variable table. In addition to these tables, there are several other tables that can be addressed, that hold auxiliary data for complex instructions, "exception ranges" (which describe the locations to which break, continue, and errors will divert control flow in a given block of code), and commands. Commands are present in order to identify places where code must be recompiled if the definition of a command changes on the fly, (for instance, if user code redefines or overloads a core Tcl command).

Most of the bytecode instructions come in multiple varieties. First, there are usually two sizes of operand: 1-byte (useful for short indices into the literal pool and the constant table) and 4-byte (for when the tables grow large enough that one byte will not suffice). In combination with these, instructions that operate on variables generally come in scalar (for when an operand is completely known at compile time – and known to be local to the procedure), array (for where an array name is known, but the array key is not), stack (for when a scalar's name cannot be determined at compile time, for instance for Tcl code like `[set $x]`), and array-stack (the most complicated case, used to access dynamically named or non-local arrays) variants. A few instructions also allow for immediate data operands as opposed to having all their data on the stack.

Control transfers take the form of jumps and conditional jumps `jumpTrue` and `jumpFalse`. The latter consume an operand from the stack and decide whether or not to jump based on whether or not its is equal to zero. There is also an `invoke` instruction that calls a Tcl command, `break`, `continue` and `done` instructions that escape an inner context to an outer one (using the exception range table to determine where to go), and a set of operations for returning and throwing errors.

Most if not all of the math operations and built-in functions in expressions have bytecode instructions that support them; all of these instructions work by consuming some number of operands from the stack and producing a single result.

In addition, there are highly complex operations that manage things like `[foreach]` loops and `[dict update]` blocks. These instructions frequently reference elaborate data structures stored in the "auxiliary data" arrays.

All told, there are about 150 different instructions in the bytecode engine. More get added with every Tcl release.

For a more comprehensive explanation of Tcl bytecode execution, see [LEWI96].

3 How compilation works.

Tcl scripts are compiled as late as possible: generally, for instance, a procedure body is compiled when the procedure is first invoked. A script is compiled by breaking it apart into commands (and comments, which are ignored), breaking the commands into words, and breaking the words into tokens (which may be constant strings, or variable, command or backslash substitutions). The parsing interfaces that the compiler uses are also exported at the C level from the Tcl library.

Code generation is orchestrated in a data structure called a `CompileEnv`. This structure manages a group of dynamically resizable arrays that hold the bytecodes, the literal pool, the local variable table, the exception and command ranges, and so on.

When a script is being compiled, if the compiler encounters a built-in command that has a "compilation procedure" associated with it, it invokes the compilation procedure, passing it the arguments to the command (in a `Tcl_Parse` structure) and the compilation environment. Commands such as `[if]` and `[for]` will then invoke the parser recursively to parse embedded expressions and scripts.

Because of Tcl's dynamic nature, a parse error in an individual command does not prevent the entire script from being compiled. Instead, code is inserted to invoke the uncompiled form of any command whose compilation procedure reports an error. If something changes before the direct invocation is called (for instance, the command is redefined), the direct evaluation will still succeed.

The fact that there is a fallback to direct evaluation means that not all commands' compilation procedures can handle all cases. In complicated situations, a compilation procedure can "throw up its hands," and defer the problem to run time. This technique is necessary, for instance, to handle unbraced expressions, where the correct parse cannot be determined until the results of command and variable substitution are known.

The assembler fits into this structure by defining an `[assemble]` command in the `tcl::unsupported` namespace. This command's compilation procedure emits inline code into the bytecode of the script that contains the `[assemble]` call. In the event that the assembly code cannot be parsed, the run-time command creates a fresh `CompileEnv` and tries again at run time. Only this second attempt will report an ultimate failure.

4 The assembly language

The assembler uses the same parser as Tcl itself, so the assembly language has the same fundamental syntax as Tcl. Because the assembly language is simpler, several things that can appear in a parsed Tcl script (namely, variable and command substitutions, and {*} expansion) simply are reported as errors. Comments, separation of commands with semicolons and newlines, joining of lines with terminal backslashes, and quoting all follow exactly the same rules as Tcl, so the syntax of assembly code should be familiar to a Tcl programmer.

Since the compilation environment is shared between the assembly code and the containing script, local variables and the current namespace are the same for both. The sharing of the compilation environment also allows for mixing of languages, as we shall see below.

Each 'command' in the assembly code represents a single bytecoded instruction. As a

convenience to the programmer, one source instruction may map into a choice of bytecodes, so the assembly programmer need not worry about the correct choice of operand sizes and addressing modes.

A few "operations" are provided that do not correspond to bytecode instructions. The most significant of these are `label`, which defines a symbolic name for the current instruction pointer, so that jump instructions can refer to it, and two language-mixing primitives, `eval` and `expr`.

When `eval` appears in the source text, its operand (which must be a constant string) is interpreted as a Tcl script, by calling the Tcl script compiler recursively in the same compilation environment. Of course, the script can share the same variables and appears in the same namespace as the assembly code. (The result at runtime is that nothing is consumed from the stack and the script's result is pushed to the stack.) Similarly, `expr` results in compiling a Tcl expression (again, a constant string) and pushing its result to the stack.

This language mixing means that code can be written in the language most appropriate to the problem. Where a given source notation is particularly convenient (for example, infix expressions), the programmer can simply switch languages. The power of all three notations (arithmetic expressions, Tcl scripts, and bytecode assembly) is captured in a single package. (To be fair, this idea was anticipated in the compiler for the L programming language [BONI06].)

Literals and variables are added to the local tables as they are encountered, and a symbol table is maintained of jump labels. The assembler runs in a single pass; jumps to undefined labels are emitted as 4-byte jumps (because the jump displacement is not known in advance) and the address of the jump target is filled in when the label is encountered.

5 Challenges faced with the assembler

Fitting the assembler into the Tcl compilation system, which was not designed for user extension, involved some interesting challenges. Among these were stack management and the possibility of instructions for which correct code cannot be generated.

5.1 Stack management

Tcl's execution system preallocates space for the operand stack, so that individual push and pop operations do not need to check for stack overflow. The way it does this is that each bytecode sequence is expected to assert the maximum number of operands that it can put on the stack. In assembly code, which shifts operands onto and off the stack freely interspersed with jumps, this depth can be difficult to determine.

Moreover, assembly code that unbalances the stack could crash the interpreter quite easily. For these reasons, we decided early on that the assembler will require a comprehensive analyzer for maximum stack consumption and for stack safety. It works according to the following outline:

1. It partitions the program into a series of "basic blocks": segments of code that are executed sequentially without jumps. Any appearance of a jump in a program ends the basic block in which it appears; any appearance of a label begins a new basic block, since the label is at least a potential jump target.

2. As code is generated, the checker updates the stack requirements of the basic blocks. It tracks three numbers: the high water mark (the maximum number of

stack elements that the block may have pushed), the low water mark (the minimum stack depth – which may be negative if a block of code begins by consuming operands), and the net effect (the difference between the stack depth on entry to and exit from the block).

3. After all code is generated, the checker begins at the entry to the assembly program, and visits basic blocks in a depth-first traversal of the control flow graph. Each basic block has zero, one or two successors: zero if it is a return (or, in some cases, a break, continue or error); one if it ends with an unconditional jump or execution falls off the end without jumping, and two if it ends with a conditional jump (the succeeding block, and the jump target). As it walks the graph, it uses the net effects to compute the stack depth on entry to every block other than the first.

4. If a block in the traversal is already visited, the stack depths are checked for consistency. If a block can be arrived at by two different paths through the code that have different net effect on the stack, an error is reported. If a block is not visited, its stack depth on entry is recorded and its successors are visited recursively. The low water mark is also examined, to guard against rogue assembly code underflowing the stack. The high-water mark for stack consumption is also tracked.

5. Once all blocks have been visited, low and high-water stack commitments are known. The stack depth at the end of the exit block is the net effect on the stack of the assembly code. It is enforced that assembly code, like any other Tcl command, leaves a single result on the stack.

Since stack effects are monitored closely, and since the assembler generates no unchecked memory addresses (a program, for instance, cannot access outside the bounds of the literal pool or local variable table), it may turn out that the assembler always generates “safe” code in the sense that it will not cause pointer smashes in the Tcl interpreter. Nevertheless, until we have more user experience with it, the plan is to have it remain in the `tcl::unsupported` namespace. (Translation: If it breaks, you own both pieces.)

5.2 Impossible instructions

There are several situations in which a sequence of assembly code will contain an instruction for which it is impossible to emit correct code. The first of these occurs if the [assemble] command is invoked at runtime, and an operation tries to use a local variable table slot for a previously unseen variable. The LVT is not resizable, and therefore the variable cannot be converted to a slot number. (Another case that arises is that the 'increment' instructions, puzzlingly, require the variable to be in one of the first 256 slots of the LVT because they lack 4-byte variants.) Finally, it is possible that a programmer simply inserts a reference to a namespace-qualified variable that cannot appear in the LVT.

In all of these cases, the correct thing to do is to put the variable name on the stack and then use a stack-based operation to manipulate it. The problem is that for array operations, the correct order of operations is that first the variable name and then the key should be pushed; for stores, appends and increments, there is an additional operand that must also follow the variable name. Since it isn't possible to “go

back in time" and insert the push of the variable name when it was actually needed, the next best thing is to generate code to reorder the stack to put the operands in the correct sequence even when the variable name arrives late. This will result in silently generating additional instructions, but will at least yield correct code.

We have not yet attempted to deal with impossible instructions, and instead report errors when they are encountered. (It is generally fairly simple to code around the problems by inserting the stack operations explicitly.) The automated recovery from impossible instructions may be implemented by the time you read this.

6 What's the assembler good for?

An assembler is of primary use as the backend of a compiler, either a compiler for a new language, or object-code transformations on Tcl itself (for optimization, code instrumentation for profiling and callgraph analysis, and so on. Having the assembler available within the Tcl system allows bytecode to be generated from a very-high-level language such as Tcl, rather than needing to resort to C code that is intimate with Tcl's execution system.

6.1 Performance and benchmarks

The chief reason that people have been interested in an assembler is performance. The dynamic nature of the Tcl language has made it difficult to generate really good code from it. (Too much can change between compilation and execution). The problem of radical language change on the fly substantially limited early attempts such as [ROUS95] to compile Tcl to machine code, even after heroic attempts ([ROUS97]) to use type inference to constrain the problem. Assembly code can bypass all the expensive runtime checks that are needed to ensure that compiled code remains valid even in the face of such radical changes.

Other languages, as well, can target the Tcl execution engine, and in fact, several other languages have been designed ([SAH94], [BONI06]) that constrain the dynamic nature of Tcl to achieve better performance while still targeting the same execution engine.

The folklore in the Tcl community has predicted that optimization at the bytecode level is likely to give disappointing results. The bytecodes, the pundits have said, are close enough to being precise counterparts to Tcl commands that little room for modification of the code sequences is actually available. Of course, without an assembler, there was no way to test this hypothesis. Given the assembler, we can attempt a benchmark to test how much we can improve a representative Tcl script by hand optimization.

Let's examine a Tcl script to compute Stanisław Ulam's "3n+1 function" [WEIS??]. The script is fairly simple, performing a handful of mathematical operations (albeit in a quite chaotic manner).

```
proc ulam1 {n} {
   set max $n
   while {$n != 1} {
      if {$n > $max} {
        set max $n
      }
      if {$n % 2} {
        set n [expr {3 * $n + 1}]
      } else {
        set n [expr {$n / 2}]
      }
   }
   return $max
}
```

Figure 1. on the following page gives an overview of the generated bytecode. Contrary to the expert opinion (which one of the authors [Kenny] of this paper used to share), there appear to be several opportunities to optimize it. (Some other possibilities can be found in [KENN02].)

First, we can observe that there are various bits of completely useless code. For instance, in

block L111 at the bottom of the chart, there is a sequence – in straight-line code – that simply pushes a zero onto the operand stack and pops it off again. This sequence can be removed entirely without ill effect. (There are perhaps similar cases elsewhere in the flow.)

It is tempting to consider further optimizations such as tail merging and common subexpression elimination. Let's start by examining the code flowing into block `L103` just below the centre of the diagram. Both the earlier blocks end with `store n`. It would be tempting to place a single `store n` at the head of `L103` instead.

Unfortunately, without further information about the program, it's also incorrect! The problem is that an variable trace can determine (through the error stack, if by no other means) what command caused it to fire. If there is a trace present on the variable n, then at least one execution path will present the wrong calling command to the trace.

That said, few Tcl programs ever attempt to establish traces on procedure-local variables. While it is challenging to prove that local variable traces are possible (commands could get redefined, or other traces could set them), a programmer – and certainly an assembly programmer – can make such assumptions with impunity. So let's start moving code around, beginning with that offending `store n` instruction.

When the instruction is moved as suggested, there is then a straight-line sequence:

```
store n
pop
load n
```

This is another sequence with an obvious optimization: remove the second two instructions (and add the load to the bottom of block `L0`). Popping the value that we just stored, only to load it back again, is another thing that simply wastes time.

If we continue propagating similar changes through the flowchart, we eventually realize that – if we assume there are no traces on local variables – we can simply keep their values on the stack, and use deep stack accessors to bring them to the top at need. This technique is almost precisely parallel to doing "frame pointer omission" in compilers targeting x86 hardware ([OSTE07]) in that it gains performance by keeping local references stack-relative, and that it makes debugging (either with conventional debuggers, or with Tcl's traces) much more challenging.

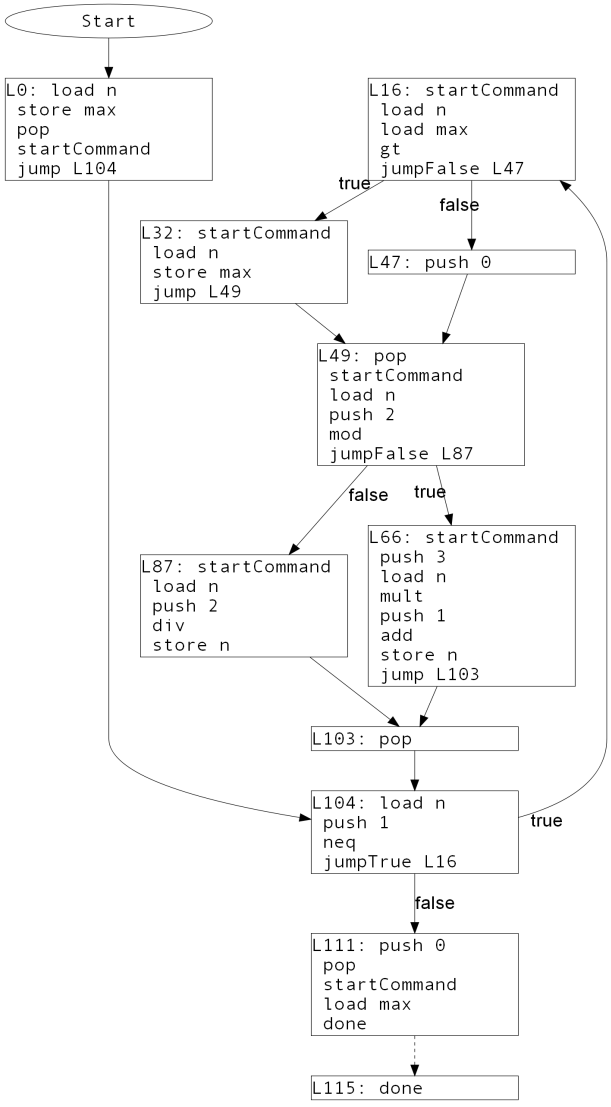

Figure 1. Flowchart of 'ulam1'

Nevertheless, let us proceed, with more enthusiasm than caution to revise the code so

that `n` is at top of stack on entry to each block, and `max` is the next on the stack. Loading `n` is then simply `dup`, while loading `max` is `over 1`.

Figure 2. shows the transformed code. The only really tricky bit is the translation of `set max $n`. Readers are invited to verify for themselves that the codeburst in block `L24` has the desired effect.

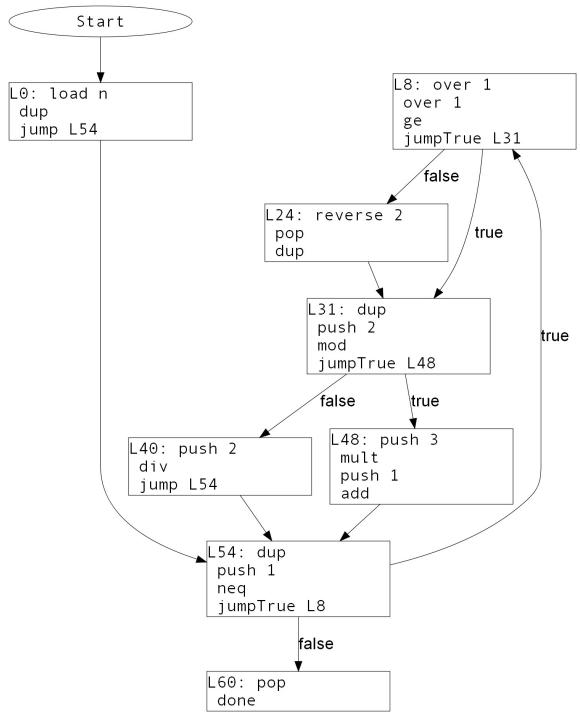

Figure 2. Hand-optimized version of 'ulam1'

What have these transformation achieved? They've roughly halved the size of the code (from 116 bytes down to 61). To see whether they make a difference in speed, we run the Tcl version and the assembly version on an unloaded 2.6 GHz Core 2 machine, and turn the two procedures loose on the first 30000 integers. For the given test case, in the best of ten runs, we find that the Tcl version takes 1.46 seconds, while the assembly version takes 1.00 seconds. While a 32% improvement in run time is not spectacular, it is surely something that could be worth pursuing for code "hot spots" that are the speed-determining steps of an application.

6.2 Targeting other languages

As mentioned earlier, making bytecode assembly available at the Tcl level makes it much easier to integrate other bytecode-targeting languages such as L or Rush. It would be an interesting experiment to see whether a language not originally designed to interoperate with Tcl could be retargeted to the Tcl virtual machine using a bytecode translation layer. Identifying missing features in the Tcl virtual machine might not only make it able to target more languages, but also give ideas how to improve its performance, stability or footprint for Tcl itself.

6.3 Bytecode rewriting

In addition to optimization, bytecodes could be rewritten to add functionality such as tracing. By replacing operators like `add` and `mult`, it might even be possible to do things like expand the semantics of mathematical operators, perhaps producing an [expr] that operates on complex numbers or other extensions beyond the integers and floating-point numbers. Tcl has a rich heritage of allowing the programmer to redefine the language to suit the needs of the moment. Rewriting object code is simply another place to insert hooks to do so.

7 Work yet to be done

The assembler described in this paper is still very much at the "proof of concept" stage. It still only emits about two-thirds of the bytecodes that the Tcl engine has (although some of remaining ones are of questionable utility, and a few exist only to support legacy bytecodes loaded with `tbcload`.) Finishing the set looks fairly straightforward.

To solve the conundrum that the "impossible operations" pose, some more powerful instructions for stack rearrangement may be needed. Adding them to the Tcl engine seems sensible, and there appears to be consensus

among the maintainers that they would be useful for other purposes.

Finally, before the object-to-object transformations mentioned in the last section can be done in earnest, the Tcl disassembler has to be modified to produce code that can be assembled again. (Until now, it's been intended for display, not for machine processing, and it's accumulated various hard-to-parse accoutrements.)

8 Conclusions

The Tcl bytecode assembler has been a very frequently-requested feature, despite the fact that the Tcl pundits have frequently pooh-poohed it as being of less benefit than programmers think. Now that it is a reality, it seems to be at least potentially worthwhile (even without hacking in the bytecode engine, 30-40% speedups can be seen by rewriting in assembler code), and it will at least now be possible for assessments of its capability to be founded in science, rather than speculation.

The experimental source code is available on a feature branch in the Tcl repository, if other investigators want to examine it. To retrieve it, the command:

```
cvs -d:pserver:anonymous@tcl.cvs.sf.net:/cvsroot/tcl checkout \
        -r dogeen-assembler-branch tcl
```

will retrieve the branched source tree (a modified Tcl 8.6 HEAD). User documentation, when prepared, will also be placed in that source tree.

References

[BONI06] Bonilla, Oscar; Tim Daly, Jr. and Larry McVoy. "The L programming language: or, Tcl for C programmers." *Proc. 13th Ann. Tcl/Tk Conf.*. Naperville, IL: Tcl Association, October 2006. <http://www.bitmover.com/lm/papers/l.pdf>

[KENN02] Kenny, Kevin B.; Miguel Sofer and Jeffrey Hobbs. "Tcl bytecode optimization: some experiences." *Proc. 9th Ann. Tcl/Tk Conf.*. Vancouver, British Columbia, Canada: ActiveState, September 2002. <http://www.tcl.tk/community/tcl2002/archive/Tcl2002papers/kenny-bytecode/paperKBK.pdf>

[LEWI96] Lewis, Brian T. "An on-the-fly bytecode compiler for Tcl." *Proc. 4th Annual Tcl/Tk Conference*. Monterey, California: USENIX, July 1996. <http://www.usenix.org/publications/library/proceedings/tcl96/full_papers/lewis/>

[OSTE07] Osterman, Larry. "FPO." *Larry Osterman's Weblog*. Redmond, WA: Microsoft, March 2007. <http://blogs.msdn.com/b/larryosterman/archive/2007/03/12/fpo.aspx>

[ROUS95] Rouse, Forest R. and Wayne Christopher. "A Tcl to C compiler." *Proc. 3rd Ann. Tcl/Tk Conf.*. Toronto, Ontario, Canada: USENIX, July 1995. <http://www.usenix.org/publications/library/proceedings/tcl95/full_papers/rouse.ps>

[ROUS97] Rouse, Forest R. and Wayne Christopher. "A typing system for a multiple-backend Tcl compiler." *Proc. 5th Annual Tcl/Tk Workshop*. Boston, Mass.: USENIX, July 1997. <http://www.usenix.org/publications/library/proceedings/tcl97/full_papers/rouse/rouse.pdf>

[SAH94] Sah, Adam; Jon Blow; and Brian Dennis. "An introduction to the Rush language." *Proc. 2nd Tcl/Tk Workshop*. New Orleans, LA: Computerized Processes Unlimited, May 1994. <number-none.com/blow/papers/rush_tcl94.pdf>

[WEIS??] Weisstein, Eric. "Collatz problem." *MathWorld*. : A Wolfram Resource, .

<http://mathworld.wolfram.com/CollatzProblem.html>