An assembler for Tcl bytecode:
????

Kevin Kenny
GE Research
Ozgur Dogan Ugurlu
University of San Francisco

An assembler for Tcl bytecode:
A technological dead end

Kevin Kenny
GE Research
Ozgur Dogan Ugurlu
University of San Francisco

~~An assembler for Tcl bytecode:~~
~~a technological dead end~~

Why the Tcl maintainers can have it

(and you can't!)

Kevin Kenny
GE Research
Ozgur Dogan Ugurlu
University of San Francisco

An assembler for Tcl bytecode:
~~A technological dead end~~
~~Why the Tcl maintainers can have it~~
~~(and you can't!)~~
Opening the bytecode engine to Tcl scripts

Kevin Kenny
GE Research
Ozgur Dogan Ugurlu
University of San Francisco

Tcl Assembly Language: what it is

- A way for scripts to include instructions at the Tcl virtual machine level
- A bookkeeping system for code generation
- A way for compilers (for instance “little languages”) to target the Tcl bytecode engine
- A rapid prototyping system for Tcl code generation
- A new backend connection for L?

TAL: What it isn't

- Not a good way to chase performance in user scripts
  - C is faster and likely easier to maintain
- Not “officially supported”
  - `tcl::unsupported` namespace
  - We don't want to freeze the virtual machine
  - But migration paths are likely (tbcload)
- Not a way to extend the engine
  - Current instructions only

Still with me?

Not scared off yet?

Really?
It gets dangerous ahead...

Anatomy of a bytecode object

- Bytecode instructions themselves
- Literal table
- Local variable table
- Command table
- Exception ranges
- Auxiliary data

Bytecode instructions

- Interpreter is a stack machine like Forth or PostScript
- Most instructions work with operands on top of stack
- Instructions also have parameters stored inline in the bytecode
- Many instructions refer to offsets in the other tables (literals, local variables, etc.)
- Stack mistakes get SEGV and `Tcl_Panic`

Assembly syntax is Tcl syntax

- The assembler uses the Tcl parser
- Commands are instructions (plus a handful of "assembly directives")
- No `$`-, `[]`-, or `{*}`- substitutions.
- Very little additional stuff beyond the instructions
  - Because the assembler can figure it out.

Constants

`push` (8- or 32-bit offset into literal table)

Literal table is simply an array of `Tcl_Obj` pointers

Assembler manages literals for the programmer:

```
assemble {
    push puts
    push {hello, world!}
    invoke 2
    pop
}
```

Variables

- Instructions come in 4 basic flavors
  - Local scalar (1 or 4-byte local variable table [LVT] index)
  - Local array (LVT index plus key from the stack)
  - General scalar (name on the stack)
  - General array (name and key on the stack separately)
- Some instructions also have 'immediate' variants
- Load, store, append, lappend, incr, exist, unset
- Upvar, nsupvar, variable
- Assembler manages LVT

Variables

```
push 2
store x;       # set x 2
pop
load x
load x
add
store y;       # set y [expr {$x + $x}]
pop
push ::result
load y
storeStk;      # set ::result $y
pop
```

Operations

- Consume operands off stack and stack the result
- Lots of these:
  `add, sub, mul, div, expon, mod, neg,`
  `le, lt, ge, gt, eq, ne,`
  `bitand, bitor, bitxor, bitnot,`
  `land, lor, lnot,`
  `strmatch, strcmp, streq, strneq, ...`

Stack manipulation

- 'nop', 'pop', 'dup', 'over', 'reverse', …
- Rearrange objects on the stack
- Sometimes important that objects get accessed in the right order
  - Traces

Jumps

- Jump, jumpTrue, jumpFalse
  - 8 or 32-bit byte offset (all jumps are relative)
- Label
  - Gives a name to a jump target
  - Assembler manages relative jumps

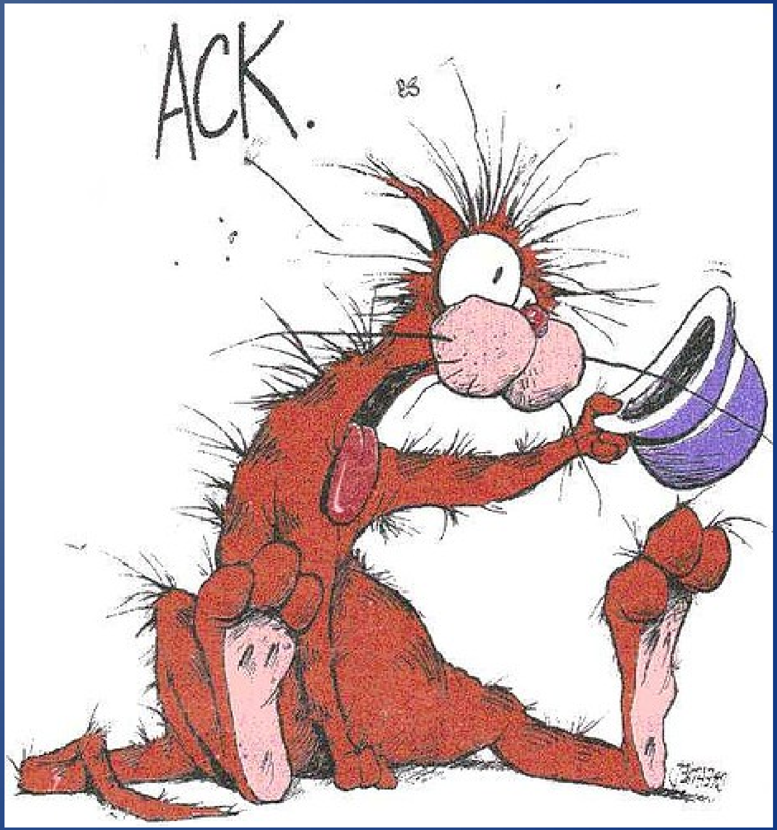
```
# set y [expr \
        {$x ? 0 : 1}]

  load x
  jumpTrue here
  push 1
  jump there
label here
  push 0
label there
  store y
  pop
```

Our first problem: Ack! A stack attack!

- Bytecode objects must know their stack consumption in advance.

```
label loop
  push 1
  jump loop
```

- Tcl_Panic!



Rules of the road for stack usage

- Assembler tracks stack depth at each instruction
- No instruction may underflow the stack
- All code paths to a given instruction must enter at the same stack depth
- Stack depth must be 1 on exit
- High water mark is calculated
- Result: *Assembly code can't smash the stack.*

Errors

- Exception ranges: “All code from bytes M to N should transfer to byte P on an exception”
- Exception ranges are nested.
- 'beginCatch' and 'endCatch' instructions mark and rollback the stack.
- 'startCatch' and 'doneCatch' directives (no code generated) make the exception range.
- Assembler again follows the control flow and checks consistency.

More stuff

- List and dictionary operations
- String match
- Regexp
- PushReturnCode, pushReturnOptions, pushResult
- ...

Invoking the interpreter

- 'invoke' – pops objv from the stack and pushes the command result.
- 'evalStk' and 'exprStk' – evaluate an object from the stack.
- 'eval' and 'expr' – invoke the compiler recursively, compiling a script or expression in line.

This wasn't very hard. I LOVE TCL!

Now that you're all
TAL programmers...

The assembler and performance

- Simple benchmark:

```
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
}
```

Squeezing the assembly code

- Move variables to the stack
  - Loses traces
  - How important is this?
- Store/pop/load optimization
- Branch-to-branch elimination
- Result: Assembly code ran in about 60% of the time.

So why bother?

- C is still 30× faster than bytecode
- And more readable
- But: The tradeoffs are different if you're a compiler writer.
- Or if most of your calculations are Tcl_Obj-oriented
- Or if you're implementing Tcl in Tcl.
- Or if you're a Core maintainer...
  … so this isn't really a dead end. It's a jumping-off point.

What's not done?

- A few families of instructions
  - 'foreach' and 'return' (Simple Matter Of Programming)
  - 'break' and 'continue' – Bugs in the compiler/engine!
  - 'dict update' and 'dict for'
  - 'expand' (does {*})
  - JumpTable
  - 'startCommand' and 'syntax' – may not be worth messing with.
- The Great Big Manual (100+ instructions to document!)

Why unsupported?

- At first, because we thought that badly written assembler code would crash the VM.
  - But we're tight about checking things. Assembler code can't do wild jumps, smash the operand or exception stacks, access off the end of tables, …
  - Should be as safe as Tcl
- But – plans afoot every so often to change the VM.
  - Would make assembly code instantly obsolete
  - Would also make TDK-compiled modules obsolete

Thank You!

And thanks to
Google Summer of Code 2010
for sponsoring this project