Jim Tcl

A Small Footprint Tcl Implementation

Steve Bennett

WorkWare Systems
What is Jim Tcl?

- Another C Implementation of the Tcl language (~Tcl 8.6)
- Small, modular
- Does not focus on large Tcl applications, Tk
- Designed for embedded applications
- Functional programming and other enhancements
Performance

![Performance Chart]

- [for] busy loop
- fibonacci(25)
- sieve
- ary
- dynamic code
- dynamic code (list)
- upvar

Time (ms):

- 6.8.1
- 7.6p2
- jim-0.71
- 8.4.19
- 8.5.8
Performance

fibonacci(25)
Memory Usage

Heap Usage during benchmark testing

Heap (MB)

jimsh (aggressive)

jimsh

tclsh 8.5
Size

Where possible, Jim Tcl uses system (libc) features to provide a small footprint at the cost of features and/or compatibility.

<table>
<thead>
<tr>
<th>System/Configuration</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim Tcl, system regex</td>
<td>3500</td>
</tr>
<tr>
<td>Jim Tcl, built-in regex</td>
<td>9878</td>
</tr>
<tr>
<td>Jim Tcl, built-in regex + utf-8</td>
<td>9929</td>
</tr>
<tr>
<td>Tcl 8.5.8 regex</td>
<td>54892</td>
</tr>
</tbody>
</table>

regexp + regsub implementation
## Size - Modular

<table>
<thead>
<tr>
<th>Component</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td>89957</td>
</tr>
<tr>
<td>tclcompat</td>
<td>4641</td>
</tr>
<tr>
<td>load</td>
<td>633</td>
</tr>
<tr>
<td>package</td>
<td>1594</td>
</tr>
<tr>
<td>readdir</td>
<td>509</td>
</tr>
<tr>
<td>glob</td>
<td>2002</td>
</tr>
<tr>
<td>array</td>
<td>1696</td>
</tr>
<tr>
<td>clock</td>
<td>1437</td>
</tr>
<tr>
<td>exec</td>
<td>5290</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>6606</td>
</tr>
<tr>
<td>posix</td>
<td>1548</td>
</tr>
<tr>
<td>regexp</td>
<td>3508</td>
</tr>
<tr>
<td>signal</td>
<td>3689</td>
</tr>
<tr>
<td>aio</td>
<td>7629</td>
</tr>
<tr>
<td>eventloop</td>
<td>5060</td>
</tr>
<tr>
<td>pack</td>
<td>2724</td>
</tr>
<tr>
<td>binary</td>
<td>5289</td>
</tr>
<tr>
<td>utf-8</td>
<td>16563</td>
</tr>
</tbody>
</table>
History

Jim Tcl - Performance and Size over Time

- PI digits
- ary
- ary [dict]
- ary [static]
- heapsort
- mandel
- Executable Size

Add many features from TinyTcl: exec, clock, file, array, etc.

Static array optimisation

subt optimisation
Cross Compile

- Easy to cross compile on many platforms
- Select components, options with 'configure'

```
$ ./configure --ipv6 --with-ext=binary --host=arm-linux
Host System...arm-unknown-linux-gnu
Build System...x86_64-apple-darwin11.1.0
C compiler...ccache arm-linux-gcc -g -O2
C++ compiler...ccache arm-linux-c++ -g -O2
Build C compiler...cc
Checking for stdlib.h...ok
Checking for long long...ok
..etc..
$ make
```
Source Tracking

- Tcl is a very dynamic language
- Tracking source location is not trivial
- Error messages can be hard to interpret

$ tclsh8.6 dbgtest.tcl
  can't use non-numeric string as operand of "+"
  while executing
  "expr 1+$x"
    (procedure "p4" line 3)
    invoked from within
  "p4 y"
    (procedure "p2" line 4)
    invoked from within
  "p2"
    (procedure "p1" line 3)
    invoked from within
  "p1"
    (file "dbgtest.tcl" line 33)
Source Tracking

Jim Tcl gives absolute line numbers

$ jimsh dbgtest.tcl
Runtime Error: dbgtest.tcl:8: syntax error in expression: "1+y"
in procedure 'p1' called at file "dbgtest.tcl", line 33
in procedure 'p2' called at file "dbgtest.tcl", line 28
in procedure 'p4' called at file "dbgtest.tcl", line 22
at file "dbgtest.tcl", line 8

• Error messages are easier to interpret
• Source location introspection allows for parsers, debuggers, code coverage tools
Source Tracking

<table>
<thead>
<tr>
<th>Token #</th>
<th>Token Type</th>
<th>Token Value</th>
<th>Object Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>LIN</td>
<td></td>
<td>scriptline line=1</td>
</tr>
<tr>
<td>[1]</td>
<td>ESC</td>
<td>set</td>
<td>source (test.tcl:1)</td>
</tr>
<tr>
<td>[2]</td>
<td>ESC</td>
<td>x</td>
<td>source (test.tcl:1)</td>
</tr>
<tr>
<td>[3]</td>
<td>ESC</td>
<td>abc</td>
<td>source (test.tcl:1)</td>
</tr>
<tr>
<td>[4]</td>
<td>LIN</td>
<td></td>
<td>scriptline line=2</td>
</tr>
<tr>
<td>[5]</td>
<td>ESC</td>
<td>if</td>
<td>source (test.tcl:2)</td>
</tr>
<tr>
<td>[6]</td>
<td>STR</td>
<td>[string match -x* $x]</td>
<td>source (test.tcl:2)</td>
</tr>
<tr>
<td>[7]</td>
<td>STR</td>
<td>\puts &quot;$x matches\n&quot;</td>
<td>source (test.tcl:2)</td>
</tr>
<tr>
<td>[8]</td>
<td>ESC</td>
<td>else</td>
<td>source (test.tcl:4)</td>
</tr>
<tr>
<td>[9]</td>
<td>STR</td>
<td>\puts &quot;$x does not match\n&quot;</td>
<td>source (test.tcl:4)</td>
</tr>
</tbody>
</table>

1: set x abc
2: if {if {[string match -x* $x]} { 
3:   puts "$x matches"
4: } else {
5:     puts "$x does not match"
6: }

Initial parse preserves source location of each token plus the line

test.tcl
## Source Tracking

As script is evaluated, tokens are converted to internal representation.

```tcl
1: set x abc
2: if {[[string match -x* $x]]} {
3:   puts "$x matches"
4: } else {
5:   puts "$x does not match"
6: }
```

### Tokens and Type Values

<table>
<thead>
<tr>
<th>Token #</th>
<th>Token Type</th>
<th>Token Value</th>
<th>Object Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>LIN</td>
<td></td>
<td>scriptline line=1</td>
</tr>
<tr>
<td>[1]</td>
<td>ESC</td>
<td>set</td>
<td>command</td>
</tr>
<tr>
<td>[2]</td>
<td>ESC</td>
<td>x</td>
<td>variable</td>
</tr>
<tr>
<td>[3]</td>
<td>ESC</td>
<td>abc</td>
<td>source (test.tcl:1)</td>
</tr>
<tr>
<td>[4]</td>
<td>LIN</td>
<td></td>
<td>scriptline line=2</td>
</tr>
<tr>
<td>[5]</td>
<td>ESC</td>
<td>if</td>
<td>command</td>
</tr>
<tr>
<td>[6]</td>
<td>STR</td>
<td>[string match -x* $x]</td>
<td>expression</td>
</tr>
</tbody>
</table>
| [7]     | STR        | \nputs "$x matches"
     | source (test.tcl:2)        |
| [8]     | ESC        | else        | compared-string           |
| [9]     | STR        | \nputs "$x does not match"
     | script (test.tcl:4)        |
### Source Tracking

#### Source Information

Source information propagates as scripts are parsed and evaluated.

---

Tcl/Tk 2011

---

<table>
<thead>
<tr>
<th>Token #</th>
<th>Token Type</th>
<th>Token Value</th>
<th>Object Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>LIN</td>
<td>scriptline line=5</td>
<td>scriptline line=5</td>
</tr>
<tr>
<td>[1]</td>
<td>ESC</td>
<td>puts</td>
<td>source (test.tcl:5)</td>
</tr>
<tr>
<td>[2]</td>
<td>ESC</td>
<td>$x does not match</td>
<td>source (test.tcl:5)</td>
</tr>
</tbody>
</table>

---

### Example Code

```tcl
1: set x abc
2: if {[[string match -x* $x]]} {
3:   puts "$x matches"
4: } else {
5:   puts "$x does not match"
6: }
```

---

- **Test Script:**
  ```tcl
  test.tcl
  ```
  ```tcl
  puts "$x does not match"
  ```
Source Tracking

Access source information for any string

```
1: # test3.tcl
2: puts [info source {}]
3: 
4: proc a {} {
5: }
6: 
7: puts [info source [info body a]]
8: 
9: set b {
10:   one
11:   two
12:   three
13: }
14: puts [info source [lindex $b 1]]
```
Source Tracking

Access source information for any string

1: # test3.tcl
2: puts [info source {}]
3:
4: proc a {} {
5: }
6:
7: puts [info source [info body a]]
8:
9: set b {
10:   one
11:   two
12:   three
13: }
14: puts [info source [lindex $b 1]]
Source Tracking

Pure-Tcl debugger can display source location and set source-based breakpoints

```tcl
$ ./jimdb test.tcl
Jim Tcl debugger v1.0  -  Use ? for help

@ test.tcl:1 set x abc
>   1 set x abc
>     2 if {[string match -x* $x]} {
    dbg> n
=>   abc
@ test.tcl:2 if {[string match -x* $x]} ...
   1 set x abc
>     2 if {[string match -x* $x]} {
       3   puts "$x matches"
    dbg> p $x
abc
dbg> ?
s   step into   w   where
n   step over   l [loc] list source
r   step out   v   local vars
c   continue   u   up frame
p [exp] print   d   down frame
b [loc] breakpoints t [n] trace
? [cmd] help   q   quit
dbg> 1 alias
@ stdlib.tcl
   1 # Create a single word alias (proc)
   2 # e.g. alias x info exists
   3 # if {[x var]} ...
   * 4 proc alias {name args} {
       5   set prefix $args
```
Source Tracking

Source information can be used to track code coverage in pure-Tcl

```tcl
$ ./jcov testcov.tcl
a(1) = 2
a(4) = 1

1: foreach i {abc def ghi} {
3:     switch -glob -- $i {
####:      {[a-d]*} {
  2:     incr a(1)
-:     }
####:      def {
####:        incr a(2)
-:     }
####:      g*h {
####:        incr a(3)
-:     }
####:      g*i {
  1:        incr a(4)
-:     }
####:      default {
####:        incr a(5)
-:     }
-:     }
-: }
2: parray a
```
Page files are Tcl scripts

Jim Tcl parser gives accurate error messages

The µWeb compiler is a Jim Tcl script. It uses the live stack trace information to provide source-accurate error messages and also ‘info source’ to record the original source location of “scriptlets”. “scriptlets” are executed at runtime by the Jim Tcl interpreter (via Jim_Eval_Named), Runtime errors can therefore provide accurate source information.
The µWeb compiler is a Jim Tcl script. It uses the live stack trace information to provide source-accurate error messages and also ‘info source’ to record the original source location of “scriptlets”.

Page files are Tcl scripts parsed as a DSL. They include “scriptlets” which are executed at runtime.

```
static const struct elem_button_t elem15[] = {
    ...
    .submit_script.script = "\n" "cgi success \"Message log cleared\"\n" "file delete /var/log/messages\n" \n",
    .submit_script.filename = "syslog.page",
    .submit_script.line = 41,
};
```

“scriptlets” are executed at runtime by the Jim Tcl interpreter via Jim_Eval_Named(). Runtime errors can therefore provide accurate source information.

Source information of embedded Tcl Scripts is preserved within the generated C code.

The Jim Tcl interpreter for the target platform is linked into the application.
The µWeb compiler is a Jim Tcl script. It uses the live stack trace information to provide source-accurate error messages and also ‘info source’ to record the original source location of “scriptlets”.

Page files are Tcl scripts parsed as a DSL. They include “scriptlets” which are executed at runtime.

```
37: button clear {  
38:   label "Clear Log"  
39:   help "Clear the log display"  
40:   editmode newline  
41:   submit -tcl {  
42:     cgi success "Message log cleared"  
43:     file delete /var/log/messages  
44:   }  
45: }
```

“scriptlets” are executed at runtime by the Jim Tcl interpreter via Jim_Eval_Named(). Runtime errors can therefore provide accurate source information.
Open On-Chip Debugger
Free and Open On-Chip Debugging, In-System Programming and Boundary-Scan Testing

• Jim Tcl as configuration, commands
• Provides a full-featured, well-known language
• Easy build integration, cross compilation

```tcl
jtag newtap $_CHIPNAME cpu -irlen 4 -ircapture 0x1 \ 
-irmask 0xf -expected-id $_CPUTAPID
set $_TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME arm7tdmi -endian $_ENDIAN \ 
-chain-position $_TARGETNAME -variant arm7tdmi

$_TARGETNAME configure -event reset-start {
    # start off real slow when we're running off internal RC oscillator
    jtag_khz 32
}
proc peek32 {address} {
    mem2array t 32 $address 1
    return $t(0)
}

# Wait for an expression to be true with a timeout
proc wait_state {expression} {
    for {set i 0} {$i < 1000} {incr i} {
        if {[uplevel 1 $expression] == 0} {
            return
        }
    }
    return -code error "Timed out"
}
```
autoconf replacement
"Tcl 8.5.8 configure is 20162 lines long"

Written in Tcl

Tcl8.5 or Jim Tcl

Includes bootstrap Jim Tcl, single source file Tcl interpreter

Used by Fossil SCM, Jim Tcl
Lambdas, Garbage Collection and more

Jim Tcl allows procs with static variables

```
proc a {x} {{adder 5}} {
  return [incr x $adder]
}
a 3
8
```

And garbage-collected references

```
set r [ref "One String" test]
<reference.<test__>.00000000000000000000>
```

With finalizers (destructors)

```
finalize $r myfinalizer
.set r ""
collect
myfinalizer called with <reference.<test__>.0000000000 123
```
Lambdas, Garbage Collection and more

Which allows garbage-collected lambdas

```tcl
# Implementation of lambda with Jim Tcl
proc lambda {arglist args} {
    set name [ref {}] func lambda.finalizer
    tailcall proc $name $arglist {*}$args
}
proc lambda.finalizer {name val} {
    rename $name {}
}
```

Which can be used like any other command

```tcl
.set list {1 50 20 -4 2}
1 50 20 -4 2
.lsor - command [lambda {a b} {expr {$a - $b}}] $list
-4 1 2 20 50
```
Lambdas, Garbage Collection and more

Lambdas can include static variables, thus creating closures

```tcl
proc make-adder {x} {
    lambda p x {
        incr p $x
    }
}
set add5 [make-adder 5]
$add5 10
15
$add5 3
8
```
Jim Tcl Unique Features

- array/dict/list conversion
- built-in line editing
- modular, optional utf-8
- object-oriented I/O
- garbage collected references
- proc "static" variables
- accurate source tracking
- signal handling

- stacking local procs, upcall
- 64 bit integers
- expr shorthand: $(...)
- simplified packaging system
- proc &upvar
- proc default args in any order, rename args
- udp, IPV6, unix domain sockets, pipes
http://jim.berlios.de/