

# Jim Tcl

## A Small Footprint Tcl Implementation

Steve Bennett,

WorkWare Systems

<http://www.workware.net.au/>

[steveb@workware.net.au](mailto:steveb@workware.net.au)

July 2011

### Abstract

*Jim Tcl is a modern implementation of Tcl, designed to be small, modular, easy to build and easy to embed. Along with a high degree of compatibility with Tcl 8.5, Jim Tcl includes a number of innovative features such as lambdas, garbage collection, object-oriented I/O and signal handling. This paper presents a detailed look at some of the most interesting aspects of Jim Tcl.*

The [Jim Tcl](#) [1] project was begun in 2005 by Salvatore Sanfilippo, largely as a testbed for new features such as functional programming support which were difficult to retrofit to Tcl<sup>1</sup> and required some practical experimentation. Since then, Jim Tcl has acquired many new features, both standard Tcl features and features unique to Jim Tcl and has improved in stability and speed.

#### 1. THE STATE OF JIM TCL

Jim Tcl v0.71, which was released in June 2011;

- Runs on at least: Mac OS X, Linux (many architectures), FreeBSD, QNX, eCos, Solaris, cygwin, msys/mingw and Haiku.
- Includes many C and Tcl optional components, including: glob, telcompat, tree, rlprompt, oo, binary, load, package, readdir, array, clock, exec, file, posix, regexp, signal, aio, eventloop, pack, syslog, nvp, readline, sqlite, sqlite3, win32
- Passes over 3700 unit tests
- Is between 100KB and 220KB in size depending on selected components, platform and build options
- Has 127 built-in commands

---

Permission to make digital or hard copies of this work is granted without fee provided that copies are reproduced in full and bear this notice. To copy otherwise requires prior specific permission.

#### A Short History of Jim Tcl

The Jim Tcl project has been active for over six years.

Date	Who	Description
Feb 2005	antirez	Initial public version
Sep 2005	antirez	Enter low activity maintenance period
Jun 2008	oharboe	Take over as maintainer
Jul 2008	oharboe	Change to FreeBSD license
Nov 2008	steveb	Begin port of missing Tcl functionality
Oct 2009	oharboe	Move to git
Jul 2010	oharboe	Release 0.51
Oct 2010	steveb	Release 0.63 - Merge workware port
Jan 2011	steveb	Release 0.70 - Including utf-8 support
Jun 2011	steveb	Release 0.71

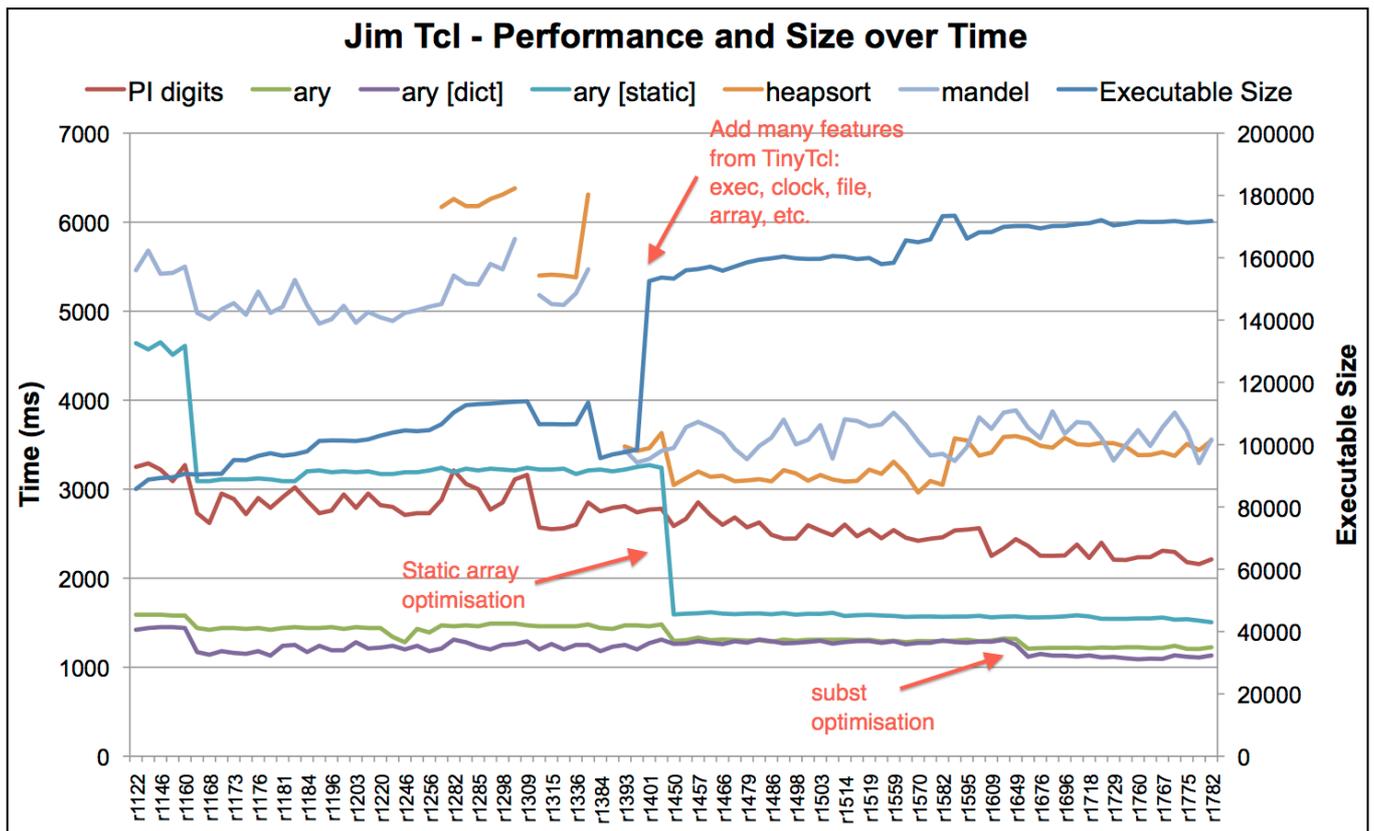
The very first publicly released version of Jim Tcl included support for references and garbage collection as well as a handful of core commands.

Subsequent releases have added many new core commands, optional extensions and significant Tcl compatibility.

The following graph shows the evolution of Jim Tcl in both size (features) and speed. A standard set of performance benchmarks is run for every, single commit to the public repository in order to monitor the size, speed and correctness over time.

---

<sup>1</sup> In this paper, the term Tcl will be used to refer to the original, official Tcl implementation — <http://tcl.sourceforge.net/> while Jim Tcl will be used to refer to the Jim Tcl implementation — <http://jim.berlios.de/>



Jim Tcl on Linux, 266MHz ARM, gcc 4.2.4 -Os <sup>2</sup>

## The Philosophy of Jim Tcl

When reimplementing an existing system, it can be difficult to balance competing goals of compatibility and whatever is driving the need for a new implementation. Jim Tcl strives to be a small footprint implementation, in both code size and memory usage, however this goal often competes with the goal of Tcl compatibility.

The philosophy of how Jim Tcl balances its goals can be summarised as:

*Jim Tcl attempts to avoid gratuitous incompatibilities with Tcl, while being open to the addition of new features which improve the usability and usefulness of Jim Tcl. Any large feature, including Tcl-compatible features, must be optional at compile time.*

The expression of this philosophy can be seen, for example, in the implementation of regular expressions (regexp, regsub) in Jim Tcl. To minimise the footprint, there are three options available (at compile time):

1. Disable regular expression support
2. Use the system-provided POSIX regex support to provide ERE<sup>3</sup> regular expression support. (This is the default)
3. Use the built-in regex support to provide (a significant subset of) ARE<sup>4</sup> regular expression support, including UTF-8.

Note that this approach necessarily leads to some differences between Jim Tcl and Tcl, and even between different configurations of Jim Tcl, but the size difference is significant.

Configuration	Size (bytes)
Jim Tcl, system regex	3500
Jim Tcl, built-in regex	9878
Jim Tcl, built-in regex + utf-8	9929
Tcl 8.5.8	54892

<sup>2</sup> Note that executable size represents the default configuration, which includes additional components over time.

<sup>3</sup> ERE — POSIX Enhanced Regular Expressions (see also BRE — Basic Regular Expressions)

<sup>4</sup> ARE — Advanced Regular Expressions

Jim Tcl does not attempt to present a stable C API. The ability to change the API from release to release allows new features to be added to Jim Tcl far more rapidly than would otherwise be the case. With the primary target for Jim Tcl being embedded scenarios, recompiling applications when upgrading to a later release is an acceptable tradeoff.

## Similarities with Tcl

Today, Jim Tcl passes several thousand test cases, most of which are fully compatible with Tcl. Jim Tcl includes support for almost all of the core Tcl commands, including: `append`, `array`, `switch`, `catch`, `break`, `continue`, `string`, `list`, `llength`, `lindex`, `lsort`, `lsearch`, `regexp`, `regsub`, `upvar`, `uplevel`, `foreach`, `dict`, `lassign`, `lset`, `exec`, `format`, `scan`, `binary` and many more. In addition Jim Tcl supports `{*}`, loadable modules, modifying the environment to `exec` via the `$env` array, binary strings, UTF-8 strings, dictionaries and `tailcall`.

Many Tcl scripts will work unchanged, especially those which avoid the use of namespaces, safe interpreters, threading, traces and, of course, Tk. Jim Tcl implements the [Dodekatalogue](#) [2].

Developers familiar with Tcl have been able to almost seamlessly make the transition to Jim Tcl.

## Missing features and capabilities

Jim Tcl omits support for a number of Tcl features, usually due to one of the following reasons.

1. The functionality has little relevance, or at least is not critical, for an embedded system or embedded application (namespaces, safe interpreters)
2. The functionality is too large and/or complex to consider adding (dynamic encodings, byte code compiler)
3. There has been no interest in the feature by someone willing to work on it (coroutines, Tk)

The following is an abbreviated list of features missing from Jim Tcl compared with Tcl 8.5:

- Namespaces
- Traces (variable traces and execution traces)
- Byte code compilation
- Safe interpreters
- Threads
- Dynamic encodings (`fconfigure -translation`, etc.)
- Tk

In addition, a number of commands omit certain options and/or subcommands, such as `lsort -dictionary -stride -unique`, `clock add`, `string wordend`, `wordstart`.

## Jim Tcl-specific features and capabilities

The Jim Tcl project started as a platform to experiment with new features, especially those related to functional programming such as closures, references, garbage collection, lambdas and tail calls. The ongoing development of Jim Tcl maintains the philosophy of pushing the boundaries when implementing new features, while still carefully considering the pros and cons with maintaining Tcl compatibility.

The following are some of the unique features of Jim Tcl, the first three of which will be explored in greater depth in the remainder of this paper.

- Functional programming support, including references, closures, lambdas and garbage collection
- Accurate tracking of source locations and source accurate error messages
- Fast, simplified packaging system
- Built-in line editing
- Procs allow default args anywhere (TIP #288)
- Procs support automatic upvar syntax: `&ref`
- Expression shorthand syntax: `$(...)`
- Procs can be stacked and invoked with `upcall`
- Signal handling
- Integers are 64-bit on supported platforms
- Supports 'jimsh -e' for immediate evaluation
- Object Oriented I/O
- Built-in support for syslog, IPv6, UDP, UNIX domain sockets and pipes on supported platforms
- Automatic conversion between list, dict and array
- Very modular with many features such as `clock`, `regexp`, `binary`, `exec`, `glob`, `package` and even I/O being optional
- Very easy to cross compile
- Single source file bootstrap jimsh can be built with just a C compiler.

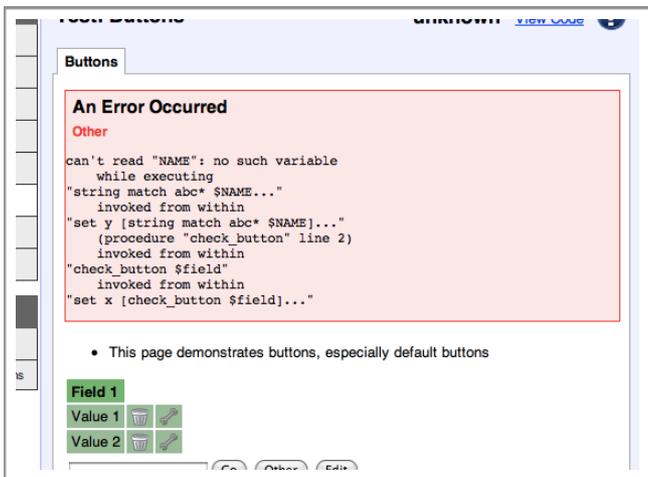
Jim Tcl is not simply a cut-down version of Tcl. Many of these features are designed to simplify code, simplify deployment and provide a very capable dynamic language, especially for embedded systems.

For example, built-in support for signal handling, UDP, UNIX domain sockets and syslog make it possible to build small, but highly capable scripts and daemons with no additional libraries or components required.

## 2. SOURCE ACCURATE ERROR MESSAGES

One of the downsides of a language as dynamic as Tcl is that it can be difficult to provide accurate source information in error messages since any string can potentially be evaluated as a script and that string could have been created in arbitrarily many ways.

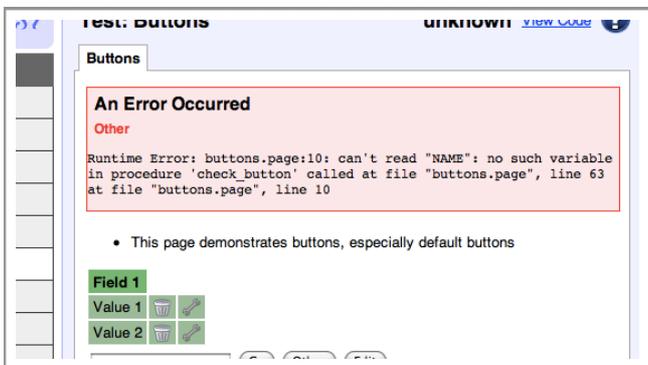
This issue was significant in our product, [μWeb](#) [3] which formerly used [TinyTcl](#) [4] (based on Tcl 6.7) as the scripting engine. While TinyTcl provides a small footprint scripting language and allowed for rapid development, it also deferred some errors until runtime. The following was the typical result of a runtime error:



μWeb with TinyTcl — runtime error message

In μWeb, Tcl scripts are defined in “page description files” from which they are parsed and embedded in C code. The stack trace as shown above describes the error, but it can be difficult to match up with the original source.

One of the most compelling reasons to move from TinyTcl to Jim Tcl was the better error reporting. Compare the same error when using Jim Tcl as the scripting engine.



μWeb with Jim Tcl — runtime error message

Notice that the exact line number is identified for each level of the stack trace, even though the original source has been parsed and embedded in C code.

```
1: title "Test: Buttons"
2: label "Buttons"
3:
4: storage none
5:
6: summary {Test submit buttons}
7:
8: init -tcl {
9:     proc check_button {name} {
10:        set y [string match abc* $NAME]
11:    }
12: }
...omitted...
59: button other {
60:     label Other
61:     editmode none
62:     submit -tcl {
63:        set x [check_button $field]
64:        cgi success "Got [cgi get text]"
65:    }
66: }
```

buttons.page

Identification of the exact location of the error makes it significantly easier for our customers, especially those new to the platform or Tcl to find and fix errors.

Below we discuss how Jim Tcl implements source tracking in such a way that it is both accurate in a highly dynamic language, and economical in resource usage.

### Accurate Source Tracking — How it Works

In Tcl versions up to approximately 8.3, `Tcl_Eval()`, the heart of the Tcl interpreter parsed and evaluated scripts for every command. A while loop with 1000 iterations re-parsed the commands in the body of the loop 1000 times. While this made the interpreter simpler and consumed less memory, it had poor performance with some scripts. Starting with Tcl 8.4 and the introduction of the byte code compiler, parsing and execution were separated, resulting in a dramatic increase in performance. While Jim Tcl eschews the complexity and size of a byte code compiler and evaluation engine, it similarly separates parsing and evaluation for a significant performance boost.

The approach to parsing scripts into an internal representation is at the heart of how Jim Tcl manages source location information, and the core structure used is the Jim “Object”, or `Jim_Obj`.

## Jim Objects

Similarly to the Tcl\_Obj structure in Tcl, Jim uses a reference counted Jim\_Obj structure to cache an appropriate internal representation for “objects” in order to improve performance. Simple internal representations are used for (64 bit) integers, floating point values and strings, while more complex internal representations are used for more complex objects such as scripts, expressions, variables and commands.

Consider the following script:

```
incr x 3
```

After parsing and evaluating, these three “words” become the following three Jim\_Obj structures:

<b>string</b>	incr
<b>type</b>	command
<b>int-rep</b>	pointer to struct Jim_Cmd
<b>string</b>	x
<b>type</b>	variable
<b>int-rep</b>	pointer to variable value plus scope info
<b>string</b>	3
<b>type</b>	int
<b>int-rep</b>	Integer 3 as a 64 bit value

While the string value is available whenever required, the internal representation acts as a cache for the most recent use of the value. For example if this command is executed in a loop, the command, variable and integer are immediately accessible without parsing or conversion.

Although this approach uses more memory than the simpler re-parsing approach, the additional memory required is modest while the performance gains are significant. It also makes it possible to associate additional information with each “word” or “token”.

The following explains how these specialised internal representations are used to carefully track source locations through the interpreter.

## Script Parsing

Consider the following simple script.

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

test.tcl

When this script is evaluated via the source command (and thus Jim\_EvalFile()), or via Jim\_EvalSource() the original source filename and line number are known. A Jim\_Obj structure is created for the script with a type of “source” and the filename and line number of the first line of the script are recorded.

<b>string</b>	set x abc\nif {[string match -x*...
<b>type</b>	source
<b>int-rep</b>	test.tcl:1

### Initial Jim\_Obj representation of the script

When this script is evaluated (which will be immediately in this case), the script is parsed and converted to a “script” object with an internal representation as follows:

<b>string</b>	set x abc\nif {[string match -x* \$x]}...
<b>type</b>	script
<b>int-rep</b>	test.tcl:1 plus script token list

### Jim\_Obj representation after conversion to script

Where the token list associated with the script is:

Token Type	string	type, int-rep
LIN		scriptline line=1
ESC	set	source (test.tcl:1)
ESC	x	source (test.tcl:1)
ESC	abc	source (test.tcl:1)
LIN		scriptline line=2
ESC	if	source (test.tcl:2)
STR	[string match -x* \$x]	source (test.tcl:2)
STR	\nputs "\$x matches"\n	source (test.tcl:2)
ESC	else	source (test.tcl:4)
STR	\nputs "\$x does not match"\n	source (test.tcl:4)

### Token list after conversion to script

Every token in the script becomes a Jim\_Obj, initially of type “source” which records the original source location of that token.

When the script is evaluated, the internal representation of each Jim\_Obj in the token list is converted as required from the “source” object.

Token Type	string	type, int-rep
LIN		scriptline line=1
ESC	set	command
ESC	x	variable
ESC	abc	source (test.tcl:1)
LIN		scriptline line=2
ESC	if	command
STR	[string match -x* \$x]	expression
STR	puts "\$x matches"	source (test.tcl:2)
ESC	else	compared-string
STR	puts "\$x does not match"	script (test.tcl:4 plus token list)

#### Token list after evaluating script

Notice how the object associated with each word of evaluated script has changed internal representation based on how it is used. Most objects have lost the original source location (each object can have only one internal representation). However any “script” objects (such as the “else” arm) retain the source location. Also the “scriptline” object for each command in the script retains the source location.

This continues for each script which is executed, where the source location in the original “source” object is propagated into the token list of the script.

#### When source tracking is not possible

Now it is possible to create situations where the source information is totally lost, or was never available. For example:

- A script which was entered via a UI element such as a GUI widget or web form (probably a bad idea!)
- A script which was read from a file without the use of ‘source’ or ‘package require’
- A script which was “composed” from strings which have no source information.

All of these scenarios are likely to be less common in practice than scripts which are executed or derived from source files. In some of these situations there is essentially nothing that can be done, however it would be possible to provide a Tcl command to set source information. Consider the following possible approach to adding source information to a string where ‘makesource’ returns a new string with the given source information added.

```
set f [open script.tcl]
set buf [$f read]
eval [makesource $buf script.tcl 1]
```

#### Tcl access to source information

In addition to providing for more informative error messages, Jim Tcl makes source information available directly to Tcl scripts through the ‘info source’ command and through the stack introspection command ‘info frame’. Consider the script:

```
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]]
```

The ‘info source’ command examines the given string (object) and returns any source information associated with that string. The above script produces:

```
$ jimsh test3.tcl
test3.tcl 2
test3.tcl 4
test3.tcl 11
```

Whenever a command is evaluated, the current source information is propagated. During proc invocation, this information is stored in the stack frame and is available via the ‘info frame’ command. The higher level commands ‘stacktrace’ and ‘stackdump’ provide access to this “live stack trace” information. The same information is used when an error occurs and the stack is unwound. When an error is caught with ‘catch’, this stack trace is available via the ‘info stacktrace’ command as well as via the ‘-errorinfo’ key in the options dictionary.

#### Case Study — μWeb

The μWeb Embedded Web Framework makes use of Jim Tcl’s ability to preserve and access source information both during parsing and at runtime as explained in the following diagram.

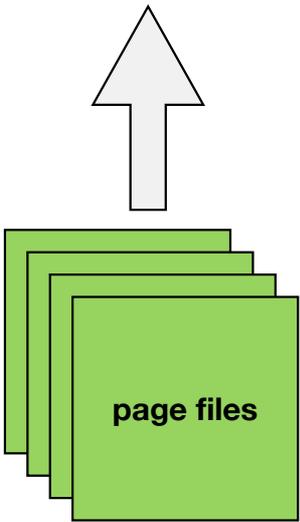
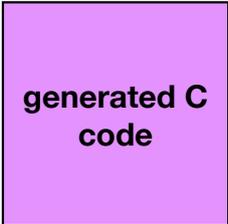
Source location is tracked from the original page definition files with Jim Tcl as a Domain Specific Language (DSL) parser, through the generated code where this information is used by the runtime Jim Tcl interpreter to produce accurate error messages which relate back to the original page definition files.

# μWeb Source Location Preservation with Jim Tcl



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".



```
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,
}
};
```

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.



## Domain Specific Language (DSL) Parser

Early versions of  $\mu$ Web used Tcl as the DSL parser. However changing to use Jim Tcl as the DSL parser had a number of benefits.

1. Supports identical Tcl-based DSL syntax
2. Error messages from the parser are more informative
3. Source location information is available for passing to the runtime interpreter
4. It is easy to ship the DSL parser as a single executable with Jim Tcl embedded.

## Source Location in the Tcl Test Framework

Jim Tcl includes a pure-Tcl implementation of `tcctest` to run the unit test suite. This implementation takes advantage of the source location information to provide the exact location of unit test failures.

```
$ ./jimsh tests/list.test
list-1.13 ERR basic tests
At      : tests/list.test:32
Expected: 'xa {} b'
Got     : 'a {} b'
-----
FAILED: 1
tests/list.test:32 list-1.13
-----
```

### The Jim Tcl version of `tcctest` provides error locations

If a test fails because of a mismatch between the result and the expected result, the location of the test body is given with 'info source'.

If a test fails because it returns an unexpected error, the location of the error is given with 'info stacktrace'.

## Experimental code coverage tool

The dynamic nature of Tcl, especially the inability to distinguish code from data can make code coverage analysis difficult. Nonetheless, a simple 50-line Jim Tcl script is able to provide useful code coverage information by simply recording the source location of every command executed.<sup>5</sup>

```
$ ./jcov test.tcl
1: set x abc
2: if {[string match -x* $x]} {
###: puts "$x matches"
-: } else {
3: puts "$x does not match"
-: }
```

### Code coverage output shows which arm was not taken

## Experimental Jim Tcl Debugger.

Although not yet available in the official Jim Tcl distribution, a pure-Tcl implementation of an interactive debugger has been developed which uses the source location information to display the source code associated with the currently executing code as well as listing source for any procedure and managing breakpoints by source location.

```
$ ./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> l 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
6     proc $name args prefix {
7         tailcall {*}$prefix {*}$args
8     }
9 }
10
11 # Creates an anonymous procedure
12 proc lambda {arglist args} {
dbg> b puts
Breakpoint at puts (tclcompat.tcl:21)
dbg>
```

### Experimental Interactive Debugger

<sup>5</sup> Both the code coverage tool and the debugger rely on an experimental command trace feature

### 3. THE JIM TCL PACKAGE SYSTEM

Tcl has a sophisticated package system for loading Tcl source and binary modules as packages. This system is also complex and potentially slow as pkgIndex.tcl files are searched and parsed.

Consider the following simple invocation.

```
$ cat pkgtest.tcl
package require blah
$ strace -e strace=open tclsh8.5 t.tcl
open("/usr/share/tcltk/tcl8.5/init.tcl",...
open("t.tcl", ...
open("/usr/share/tcltk/tclIndex", ...
open("/usr/lib/tcltk/tclIndex", ...
open("/usr/local/share/tcltk/tclIndex", ...
open("/usr/local/lib/tcltk/tclIndex", ...
open("/usr/lib/tclIndex", ...
open("/usr/share/tcltk/tcl8.5/tclIndex",...
open("/usr/share/tcltk/tcl8.5/tm.tcl", ...
open("/usr/share/tcltk/tcllib1.12/interp/
pkgIndex.tcl", ...
open("/usr/share/tcltk/tcllib1.12/png/
pkgIndex.tcl", ...
...etc..
```

**A total of 115 files are opened and read**

The need to create and deploy pkgIndex.tcl files can also be awkward.<sup>6</sup>

#### Simple Package System

With the focus of Jim Tcl on embedded environments, it is appropriate to take a much simpler approach to packaging<sup>7</sup>. The Jim Tcl packaging system:

- Has no version support. Versions are managed through filenames
- Has no index files and no autoload support
- Is fast
- Is easy to understand
- Is easy to deploy

The Jim Tcl packaging system works as follows:

1. The package subsystem maintains a list of loaded packages.
2. The command ‘package require foo’ searches each directory in \$::auto\_path for either foo.so or foo.tcl. If either file is found, the package is deemed to be located (even if loading the package fails).

3. Once the file is found, it is loaded either as a binary module or as a Tcl script.

Some notes:

1. Package names must be lower case — foo not Foo.
2. Binary loadable modules are named foo.so on all platforms.
3. The entry point for the module foo.so is Jim\_fooInit
4. Versions are expected to be handled by including the version in the name. For example ‘package require foo2’.
5. The \$:auto\_path list is initialised based on the install prefix (<prefix>/lib/jim) plus the environment variable \$JIMLIB, although applications which embed the Jim Tcl interpreter can add additional directories as appropriate.

#### Static vs dynamic packages

Jim is designed to be modular. This means both being able to omit features not required, but also making it easy to incorporate features. One example is static Tcl extensions. Pure-Tcl extensions such as glob, stdlib, telcompat and binary can easily be built as static extensions in libjim and jimsh by simply selecting them with ./configure.

```
$ ./configure --with-ext="binary glob"
```

Similarly, C-based extensions can be built either as static extensions or loadable modules.

#### External loadable extensions

Building loadable modules can be difficult on different platforms. Jim Tcl provides a helper script to make building C-based extensions as loadable modules easy on any supported platform.

```
$ build-jim-ext hello.c extra.c
Building hello.so from hello.c extra.c
Compile: hello.o
Compile: extra.o
Link:    hello.so
Test:   load hello.so
Success!
```

#### Building a loadable module is easy

The build-jim-ext script uses the configuration-time settings to invoke the compiler and linker as appropriate, including for cross compilation.

This is a “mini-TEA” [5] for Jim Tcl.

<sup>6</sup> This is not intended as a criticism of the Tcl package system, which is very powerful. Rather it explains why Jim Tcl uses a much simplified approach.

<sup>7</sup> The Jim Tcl packaging system is similar to the Tcl Module support introduced in Tcl 8.5 (<http://wiki.tcl.tk/12999>)



## Garbage Collection

Normally, all values in Tcl are passed by value. As such values are copied and released automatically as necessary. With the introduction of references, it is possible to create values whose lifetime transcend their scope.

Consider the following example where a reference is created with a finalizer.

```
. proc f {ref value} {puts "F $ref $value"}
. set r [ref 123 test f]
<reference.<test__>.0000000000
. collect
0
. set r ""
. collect
F <reference.<test__>.0000000000 123
1
```

The finalizer command 'f' is associated with the reference when it is created. (The 'collect' command is available to manually run the garbage collector, and returns the number of objects discarded. Normally the garbage collector runs automatically<sup>10</sup>.)

The first time that 'collect' is invoked, a variable 'r' exists which contains the reference. Because the reference is accessible the garbage collector has nothing to do. However the second time 'collect' is invoked, 'r' no longer contains the reference. Therefore, when the garbage collector runs it finds this dangling reference and discards it, first invoking the associated finalizer.

The finalizer is passed two arguments, the reference and the contained value, which it may use to perform any necessary cleanup.

The finalizer for a reference may be examined or changed with the 'finalize' command.

```
. finalize $r
f
. finalize $r newf
newf
```

The garbage collector works similar to the [Boehm GC algorithm](#) for C/C++ [6]. Here, the special string format makes it easy to identify strings which may be valid references. During garbage collection, the string representations of all objects are scanned for strings which could be valid references. If a given reference no longer exists in any string, the contained object is unreachable and can be collected.

## Lambda

Jim Tcl provides a lambda command which provides support for garbage collected anonymous 'functions' (Tcl procedures)<sup>11</sup> and closures.

Consider the following example.

```
. set adder [lambda a {{x 0}} {incr x $a}]
. $adder 1
1
. $adder 2
3
. set adder ""
```

An anonymous procedure is created and stored in the variable 'adder'. The procedure takes one argument which it adds to the static variable 'x' and returns the result. The procedure name '\$adder' may be used anywhere a command name is required.

The anonymous procedure is garbage collected. Once it is no longer accessible (perhaps when the procedure which defined it ends), the garbage collector is free to delete the procedure.

The implementation of the lambda command is remarkably simple.

```
# Creates an anonymous procedure
proc lambda {arglist args} {
    set name [ref {} func lambda.finalizer]
    tailcall proc $name $arglist {*} $args
}

proc lambda.finalizer {name val} {
    rename $name {}
}
```

The lambda command takes the same arguments as 'proc' except the name of the procedure is omitted. A reference is created as a unique, anonymous name for the new command. In this case the ability for the reference to contain a value is not used. The reference finalizer simply deletes the procedure. 'tailcall' is used here simply as an efficiency mechanism to avoid the creation of an additional call frame.

Lambdas can be convenient as sorting functions.

```
. set list {1 50 20 -4 2}
1 50 20 -4 2
. lsort -command [lambda {a b} {expr {$a -
$b}}] $list
-4 1 2 20 50
```

<sup>10</sup> The garbage collector runs synchronously. Whenever a new reference is created, the garbage collector will run if a certain number of references have been created or a certain period of time has passed. This means that if references are not used, garbage collection has no impact on performance.

<sup>11</sup> See [http://en.wikipedia.org/wiki/Anonymous\\_function](http://en.wikipedia.org/wiki/Anonymous_function)

## Lambda Example

The following example shows how lambdas can be useful. First note that Jim Tcl supports object-oriented I/O commands. That is, in addition to the Tcl-compatible:

```
set f [open temp.txt]
set data [read $f]
set pos [tell $f]
close $f
```

Jim Tcl supports:

```
set f [open temp.txt]
set data [$f read]
set pos [$f tell]
$f close
```

This has the advantage that it is easy to “wrap” a file handle with a procedure.

The “open |...” syntax in Jim Tcl is implemented in pure-Tcl by wrapping a file handle with a lambda.

```
1: # 'open "/..." ?mode?' will invoke
2: # this wrapper around exec/pipe
3: # Note that we return a lambda
4: # which also provides the 'pid' command
5: proc popen {cmd {mode r}} {
6:     lassign [socket pipe] r w
7:     try {
8:         if {[string match "w*" $mode]} {
9:             lappend cmd <@$r &
10:            set pids {exec *} $cmd]
11:            $r close
12:            set f $w
13:        } else {
14:            lappend cmd >@$w &
15:            set pids [exec *} $cmd]
16:            $w close
17:            set f $r
18:        }
19:        lambda {cmd args} {f pids} {
20:            if {$cmd eq "pid"} {
21:                return $pids
22:            }
23:            if {$cmd eq "close"} {
24:                $f close
25:                # And wait for the child
26:                # processes to complete
27:                foreach p $pids {os.wait $p}
28:                return
29:            }
30:            tailcall $f $cmd {*} $args
31:        }
32:    } on error {error opts} {
33:        $r close
34:        $w close
35:        error $error
36:    }
37: }
```

At line 19, a lambda is created which wraps the file handle ‘\$f’. Most subcommands are simply passed through to ‘\$f’ via the tailcall at line 30, however the new subcommand ‘pid’ is implemented at line 20 and the subcommand ‘close’ is extended at line 23.

## Jim Tcl OO

The Jim Tcl OO system uses static variables and references to implement a [pure-Tcl OO system](#) [7] with multiple inheritance in 58 lines of code.

```
$ jimsh
. package require oo
. class Account {bal 0}
. Account method deposit {x} {incr bal $x}
. Account method see {} {return $bal}
. set a [Account new {bal 100}]
<reference.<Account>.00000000000000000000>
. $a deposit 50
150
. $a deposit 25
175
. $a see
175
```

### Using the OO package

The ‘tree’ package included with Jim Tcl is largely compatible with struct::tree from tellib and is implemented as an OO class.

## 5. CONCLUSION

Jim Tcl contains many more unique features than presented here, while remaining faithful to the [Dodekatalogue](#). Tcl has seen a number of small additions over time such as {\*} list expansion, lassign, and exec redirection improvements which have made a huge difference to usability and usefulness of Tcl. Similarly, the unique features of Jim Tcl enhance its usability and facility while remaining small, fast and modular.

Not only has Jim Tcl provided a modern Tcl implementation for embedded systems, it has proven an effective platform for testing improvements to the Tcl language itself.

It is my hope that future releases of Tcl can benefit from the experience gained from implementing these improvements.

## 6. REFERENCES

- [1] <http://jim.berlios.de/>
- [2] <http://wiki.tcl.tk/10259>
- [3] <http://uweb.workware.net.au/>
- [4] <http://tinytcl.sourceforge.net/>
- [5] <http://wiki.tcl.tk/327>
- [6] [http://en.wikipedia.org/wiki/Boehm\\_garbage\\_collector](http://en.wikipedia.org/wiki/Boehm_garbage_collector)
- [7] <http://jim.berlios.de/documentation/oo/>