Static testing tools for tcl

Lindsay Marshall

Dept of Computing Science
University of Newcastle upon Tyne
UK NE1 7RU

Abstract

Simple static testing of tcl programs can detect a large class of common errors. This paper describes two tools that carry out such tests as well as providing other services.s

Introduction

The great advantage of an interpreted language like tcl is the freedom it gives you to create dynamically changing programs with ease. The great disadvantage is that simple typing errors can lurk in your code for some considerable time until they make themselves known by causing the program to fail. Tools to help find this class of problems make the life of a tcl programmer much easier. Unfortunately the dynamic nature of tcl programs makes full, *static* checking difficult, and even impossible in some cases.

However, there are simple, static checks that can pick out several of the commonest errors and this paper describes two, open source programs that carry these out as well as performing other useful tasks. I shall first discuss tclCheck, a bracket pairing tester, and then Frink which tests and formats tcl programs. In conclusion I shall look at various other checks that other programs could be written to carry out.

The programs are written to be as simple and efficient as possible so that checking programs can become a routine part of the tcl programmers development cycle. For instance, the code check−in and program release scripts I use when developing large tcl programs automatically run both programs over each module before commitment. This has proved a powerful way of eliminating some (though by no means all!) of the syntactic errors that I make most frequently.

tclCheck

Originally, tclCheck was itself written in tcl, but, in the days before byte compilation, it proved to be far too slow to meet my efficiency criteria, and so I rewrote the program in C. It reads a tcl program and prints out information about any "errors" that it finds, where an error is defined as the program finding unpaired, unescaped braces, brackets, parentheses and double quotes. The program is extremely simple - it does no detailed syntax analysis.

Tcl's rules for nesting and bracketing are rather context dependent (and at times confusing!), so tclCheck takes a conservative view of everything. Frequently it will flag things that may not in fact be errors. This generally occurs inside double quotes, so that the string ":−(" throws up an error on the (character. To stop the error you can rewrite the string as ":−\(" which is sometimes a little inconvenient, but in the long run the parenthesis pairing check usually proves to be useful enough to outweigh this. Particularly if you use regular expressions frequently. The program does try to recognise when a parenthesis might be inside in a comment as people do seem to write things like :

```
# 1) blah blah blah
```

(Though, of course, this behaviour can be turned off if necessary).

tclCheck reads through a tcl program character by character, stacking opening brackets and unstacking them when the matching closing bracket is found. By default, it will pop this stack to find a matching bracket using the precedence rule } > [> (. This means that an erroneous piece of code such as

```
{ set x [expr {a* (b+c)}}
```

would generate errors about a missing parenthesis and a missing bracket and match the pairs of braces. Experience has shown that this behaviour seems to fit the kinds of errors people make quite well, but of course it can be turned off by the use of a flag.

Two other tests that tclCheck can perform catch annoying errors that are hard to spot visually. The first is to look for lines that end with a \ followed by blank space before the end of line. In almost all cases this is an error as the \ was meant as a continuation. The other test is to look for single double quote characters that would not normally be recognised as being the start of a string. Thus it would detect the error in the statement

```
pcl p1 {$a p2}"
```

which was intended to be

```
pcl "p1 {$a p2}"
```

Sometimes it is rather hard to spot exactly where a bracket is missing even when you know that one (or more) has been omitted, so tclCheck provides a variety of "skeleton" printouts which essentially print the input program on the standard output with only leading white space and the various brackets left in. Formatting options control the look of this output which can make it much easier to identify the line where a bracket has been left out by reducing the surrounding noise.

tclCheck is fast and effective. The only possible extension that I might make to it would be to look for the matching of < and > characters, but it seems that people rarely make this error on their bind statements and the number of false positives caused by conditions probably means that the test would be more annoying than useful.

Pretty Printing

Frink, named for the sculptor Elizabeth Frink (1930 − 1993), was also first written as a pure tcl program and started out as a pretty printer for tcl. It pretty quickly became clear that the tcl version was too slow and too complicated so, like tclCheck, it was rewritten in C - I would have preferred to use C++, but at the time good C++ compilers were not universally available.

Unlike tclCheck, Frink does do some rudimentary parsing of the tcl program. It splits its input program (which is assumed to be correct tcl) into a stream of structured tokens which it then parses using a very simple, recursive descent parsing system. The program knows that every tcl statement has the format *command parameters...* and picks out complete statements for formatting. Commands whose parameter formats are known and which contain code strings are then treated specially, with the parameters that contain code themselves being parsed into further command sequences. No attempt is made to try to automatically detect code sequences in strings. It would certainly be possible to develop heuristics to do this but there would always be cases where a wrong guess was would be made.

Once a command has been broken down into its component parts, it can of course be reassembled in a variety of ways. The most obvious is the pretty printing route and this is Frink’s default behaviour. The user can control all the usual kinds of layout features such as indentation level and the presence or absence of optional keywords. Frink knows about some popular extensions such as incr tcl and tclX and can recognise and format their extended commands (though incr tcl support is not up to date with the latest version). It also knows about some popular layout styles and can reproduce these automatically.

There are two difficult problems faced when pretty printing tcl. The first, as with all code formatters, is the handling of comments. Given a completely unformatted program with long comments it is almost impossible to "do the right thing". Particularly hard, given the simple parsing strategy adopted, are end of line comments and Frink really only handles these well when they obey certain (reasonable) coding conventions. Single line comments it tends to leave alone where possible - since most people format their code as they write it and use formatters to tidy up any details this strategy usually works perfectly well.

The other difficult task is dealing with "incomplete" code fragments, that is, code created on the fly using variable substitutions etc. that arise when creating bindings or button command strings and using eval. Frequently these simply cannot be automatically formatted in any reasonable way and the program provides options which can turn off the processing of particular commands where this happens most frequently. (Of course, *any* code fragment in tcl can be incomplete in this way, but, luckily, most programmers do not make use of this feature very often.) There is an unfortunate danger of the formatter altering the input program in bad ways when this happens but this seems to be unavoidable in a static checking system.

A similar, though less common, problem occurs where tcl programmers (or extension packages) redefine the commands that Frink "knows" about. In this case it can try to format as code strings that do not contain code −- once again the solution is to turn

off special processing for these commands.

The astonishing thing is that Frink, simple as it is, manages correctly to format the vast majority of tcl programs. However, unlike tclCheck it has to be continually updated as new, code bearing features such as namespaces and interpreters) are added to the language.

Minimisation, Obfuscation and Optimisation

Once you have broken a program down into its component parts it is rather easy to consider ways of formatting other than pretty printing. In tcl 7.* days and earlier, the more characters that your tcl program contained, the slower it ran as they all had to be reprocessed every time a block of code was executed. It was a simple change to Frink to make it eliminate all redundant characters from the input program so as to reduce it to the minimum necessary to run correctly. This involves removing comments, white space and redundant {} pairs. Speed ups of 33% were reported for some programs that had been treated in this way. However with the current 8.* interpreters this feature is now pretty well redundant.

One of the features of interpreted languages that some programmers do not like is that you have to give potential users your source code which they can then read and "steal". The real solution is, naturally, to write open source code, but sometimes it may be useful to make the code harder to read. The minimisation process described above, whilst removing comments and white space, does not reduce the readability of programs as much as you might think, so Frink provides an obfuscation feature which rewrites the code in a "dense" fashion, packing multiple statements on to each line. This does produce pretty unreadable code, but, unfortunately, anyone with a copy of Frink can turn this unreadable code back into structured code rather easily, so the feature is not that useful. More comprehensive obfuscation such as rewriting variable names etc., is made impossible by the use of dynamic code creation: picking out names from strings is fraught with difficulty! It would be possible to make appropriate changes in code that is flagged as "safe" to rewrite, but the effort involved seems not to be worthwhile.

Frink also featured some experimental code optimisation features, which have now mostly been removed as the introduction of byte compilation made them redundant. These optimisations addressed the issue of string comparison which was particularly slow in tcl 7.* and earlier. Thus the simple statement

```
if {$a == {}} { }
```

could be written

```
if {[string compare $a {}] == 0} { }
```

or as

```
if {[string match {} $a]} { }
```

both of which were faster. However in all cases involving simple tests it turns out that

the using the switch statement was very much faster than any other method. Thus the above statement code be written as

```
switch {} $a { }
```

or its complement as

```
switch {} $a {} default { }
```

(The null string is placed first to eliminate the need for a −− to indicate the end of options) I tested a simple extension to Frink that could recognise if statements of this kind and rewrite them as switches. It worked well enough, but tcl 8.* improved the speed of condition testing sufficiently that the rewriting was redundant (the switch method is still faster, although now by a much smaller margin). Also, as I tend to write the switch statements explictly anyway, the extension was of no use to me personally, so the feature has been removed from the latest release of the program.

Other Checks

Since Frink carries out a simple syntactic analysis of the program it can conveniently test for several other errors that sometimes occur, such as the wrong number of parameters to a switch or extra code after the else part of an if statement. These simple tests prove to be pretty useful, especially where brackets match up but are nested incorrectly; something that tclCheck cannot detect.

Given the basic parsing structure of Frink it would be perfectly possible to add other checks to the program, though it might be preferable to make a completely new program based on it rather than further complicating the existing system. What kinds of tests would be useful for tcl programmers? This probably varies from programmer to programmer as we each make different types of error, but here are some suggestions of what I think could be useful:

- checks on variable usage: used before set, passing value to reference (append, lappend), global variable name checking
- cross reference listings for variables and procedures
- user declared proc parameter count checking
- regular expression and format string checking.
- binding event type checking − errors here sometimes only show up as a lack of a feature in the user interface rather than as an actual failure.
- option checking − looking for typos in command options that have known value sets
- namespace usage checks.

- heuristic checks for common errors

Conclusions

Simple static checking of tcl programs can detect a large number of the common errors made by programmers. The two tools described above carry out checks efficiently and effectively and so can improve productivity when writing and testing programs. However, there are error classes that these tools do not address which could easily be checked for and it would perhaps be useful to develop a suite of programs, each of which tackled one specific area. Lint lead a very successful life amongst C programmers for many years, and there is no reason why such a facility should not be built for tcl users, provided that a useful set of heuristics can be developed that can detect typical programmer clumsiness!