Brush: A New Tcl-like Language

Andy Goth
http://andy.junkdrome.org/
andrew.m.goth@gmail.com
November 2012[1]

This paper proposes a new programming language similar in structure to Tcl[2]. The new language embeds several functional programming concepts which are currently awkward or difficult to realize in Tcl. A reference system serves as the foundation for several other new features. The Tcl object model is refined to address performance concerns encountered in the Wibble[3] project. Syntax is fine-tuned to streamline preferred usage. Despite similarity to Tcl, backwards compatibility is broken. The paper discusses the motivation, design, and semantics of the language and provides a preliminary specification. Included code samples contrast how various tasks are done in Tcl and the new language.

1 This is version 3 of this document, released 20 November 2012.
2 **http://wiki.tcl.tk/tcl** Index of information on the Tcl programming language.
3 **http://wiki.tcl.tk/wibble** Wibble is a small, pure-Tcl web server.

Introduction

Tcl is a vastly powerful scripting language which blends deceptive simplicity with surprising functionality. Its minimal nature inspires programmer creativity, encouraging innovative designs difficult to imagine in more regimented languages. However, years of experience with Tcl have revealed its limitations, rough edges, and pitfalls which, while by no means fatal, could stand to be improved in order to empower programmers both experienced and new.

In the Cloverfield[4] project, Frédéric Bonnet[5] and I investigated possibilities for the next generation of Tcl. Though we shared many goals, we reached incompatible conclusions about the nature of the new language and decided to separately pursue our designs. As such, I cannot claim the Cloverfield name. For the time being, I will call my new language "**Brush**".

Brush has four primary design goals:

- Everything is a string.
- Streamline best practices.
- Enhance data structure access.
- Facilitate functional programming.

Brush does not attempt to maintain complete backward compatibility with Tcl. Several of its changes break existing scripts by subtly altering existing syntax and by defining new syntax using symbols and constructs not currently reserved.

This paper assumes a strong familiarity with Tcl script programming and a basic understanding of how Tcl is implemented.

At time of writing, Brush is **proposed but not implemented or formally specified.**

Everything is a String

One of Tcl's defining characteristics is its "everything is a string" philosophy, EIAS[6] for short. Brush embraces the Tcl EIAS philosophy without modification. EIAS informs the design of all new features in Brush. During previous iterations of Brush I considered and rejected features that conflicted with EIAS, and I redesigned others until they complied with this fundamental goal.

From the perspective of the programmer, EIAS means exactly what it says: Every value accessible within the Tcl interpreter is a string of text characters. This design offers a refreshingly natural way to program; the data is formatted in the same way it is described.

4 http://wiki.tcl.tk/cloverfield Cloverfield seeks to improve Tcl syntax and add missing features.

5 http://wiki.tcl.tk/fb Tcl Wiki page for Frédéric Bonnet.

6 http://wiki.tcl.tk/eias Discussion of the "everything is a string" concept and implementation.

EIAS offers many benefits:

- *Trivial serialization.* Strings are always suitable for transmission and archival.
- *Maximal compatibility.* Strings are a useful common denominator for all data types.
- *Easy introspection.* Displaying strings is effortless.

When a string follows a defined format, it can be said to be of the type described by that format. The important thing to notice about this definition is its fluidity: Types may exist within the imagination of the programmer, and strings can freely conform to many formats (therefore types) simultaneously. There is no need to declare types within the script. Typing is not an explicit activity; it is merely a convention, and type conventions may coexist.

Within a typical script, many types can be found: number, list, dictionary, script, math expression, regular expression, and of course string. These types all have distinct internal representations, as discussed below, but they are by no means the only types that can exist. Scripts may freely define new types built on top of existing ones; these are application-specific data structures. The definition is purely implicit in how the script chooses to access the data.

This is known as duck typing. If something walks like a duck and quacks like a duck, then it's a duck, even though it may also be a bird or a robot or a cartoon or a paranormal manifestation. Similarly, if values can be added together, they are numbers, though they are also valid as strings, byte arrays, regular expressions, and single-element lists.

Representations and Shimmering

Tcl and Brush use dual-ported values, meaning that at the C level they cache string and internal representations to avoid incessant conversions. Whenever a given representation is needed, it is either retrieved from the cache or generated from the other representation. For example, the first time a numeric value is printed, a string of ASCII digits `0-9` is generated and cached in preparation for the next time it is needed.

The dual representation design is invisible at the script level, except in rare usage patterns which induce a phenomenon known as shimmering[7]. Shimmering means frequently discarding and regenerating the string and internal representations, and it can sharply reduce performance, though it has no impact on correctness.

Shimmering becomes possible when the script repeatedly alternates its interpretation of a value's type. If the script switches between treating a value as a string and (say) a list, each time it modifies the value using one interpretation it invalidates the other cached representation, which must later be regenerated. Shimmering is possible even with purely read-only operations: When the script cycles between two or more non-string representations, e.g. between list and dictionary, each access displaces the previous internal representation since there is room for only one at a time.

7 http://wiki.tcl.tk/shimmering Shimmering is repetitive changing of the internal representation for some data.

Dict/List Unification

That last point touches on the first change Brush proposes to make to Tcl. No, Brush does not seek to add more internal representation cache slots; Brush retains the existing `Tcl_Obj`[8] structure which Tcl uses to store values. Brush's change is to unify the internal representations of lists and dictionaries. This avoids shimmering in the case of cycling between treating a value as a list and a dictionary, which is a legitimate use case experienced by the Wibble web server.

Wibble converts HTTP headers, queries, and POSTs to alternating key/value lists, which would be dictionaries if not for the possibility of duplicate keys. Tcl's [`dict`] dictionary commands ignore duplicates, so they are convenient when the website code does not expect duplicates, but in situations where duplicates are valid, the website code can instead use list-oriented commands such as [`foreach`].

EIAS requires a string representation to be generated when making a dictionary from a pure key/value list containing duplicates, since having only the dictionary representation would cause the duplicates to be lost. This duplication is very wasteful in the case of HTTP POSTs which may be megabytes in size. Having two representations doubles memory utilization.

Even in the case of canonical dictionaries (i.e. no duplicate keys), EIAS preservation incurs a performance penalty. Dictionaries are internally represented as hash tables for constant-time access, but standard hash tables are unordered, so converting a string to a dictionary and back scrambles the element order, thereby violating EIAS and the transparency of dual-ported values. Early versions of [`dict`] indeed had this problem. To correct it, the dictionary maintains a doubly linked list[9] chaining together the hash table entries in insertion order.

Tcl's implementation of dictionary values employs both hash tables and lists, yet it is not compatible with the implementation of list values. As mentioned before, alternating between [`dict`] and list commands results in shimmering, even when no write accesses are being done. It would make sense to combine the two to avoid shimmering and duplicate representations in the event of duplicate keys.

Nature of the Combined Type

The combined type is a list with an optional hash table index. The list component works just like current Tcl lists[10]: it is a linearly allocated array of `Tcl_Obj` pointers. The one change to the list structure is the addition of a pointer to an indexing hash table; if there is no index, this pointer is `NULL`. The hash table maps from keys (which, like everything else, are strings) to the indices of their corresponding values within the backing list. In this way, elements can be located in constant time either by index or by key.

8 http://wiki.tcl.tk/tcl_obj `Tcl_Obj` is the underlying data structure for all Tcl values.

9 http://core.tcl.tk/tcl/artifact/e62fccab713b5753edf37486f0947fa76bea265a Search for "`struct ChainEntry`".

10 http://core.tcl.tk/tcl/artifact/b3feb25636989d0d7d6b98f7814c01fac7e41b42 Search for "`struct List`".

In current Tcl, performing dictionary operations on a value causes its dictionary representation to be created if it does not already exist. In Brush, dictionary operations create a list representation (if needed) and construct a hash table index (if needed). One advantage is not having to discard an existing list representation; it is only necessary to augment it. Read-only list operations on a dictionary value do not need to do anything special, since it is also a list value. Using list operations to modify a dictionary value causes the index to be discarded, but the list representation need not be regenerated.

If a list value has duplicate keys, and its value is read via dictionary commands, the hash table index will simply omit all but the last of each duplicate. This results in the hash table index containing fewer than half as many elements as the list. (For canonical dictionaries, there is exactly one hash table index element for each pair of list elements.) Using dictionary commands to modify a non-canonical dictionary results in the duplicates being stripped; duplicates can be identified by not being present in the hash table index.

One drawback worth noting: Because the `Tcl_Obj` pointers are stored in a linear array, dictionary element removal takes linear time in Brush, as opposed to constant time in Tcl.

New Capabilities

Not all applications require dictionary keys to have values. Sometimes all that matters is whether or not a key is in the dictionary. This kind of data structure is called a set.

The usual Tcl implementation of a set is to make the keys map to dummy values, for example empty string. Lookup, insert, and delete all take constant time to complete, so the performance beats unsorted lists (linear time) and sorted lists (logarithmic time). The downside is that the string representation is littered with dummy values.

If the hash table index structure permits variable stride between key elements, it becomes possible to create a high-performance set structure with no need for dummy values. When the stride is two, the structure is a traditional key/value dictionary. With unit stride, the structure is a set.

Brush permits these two options, with the choice determined by which command is used to access the value. For key/value access, the traditional [`dict`] command is used. For sets, the new [`lot`] command is used.

A note on the name: [`set`] is taken, so I chose [`lot`] because it is a synonym of set and has similar pronunciation and spelling. Other possible names include: [`ring`], [`field`], [`group`], and [`corpus`], but I prefer [`lot`] because it does not abuse existing terminology. I am open to suggestions on the name.

Using a linear array to back a dictionary means that each element has a numeric index corresponding to insertion order. Classical sets are not ordered, but this feature is there in case a script needs it.

As with dictionaries, lots (sets) cannot contain duplicate keys, even though their backing lists can. Accessing a noncanonical lot with [lot] subcommands (described below) results in the duplicate keys being ignored, but they are visible to the list commands and are in the string representation. Any command that creates or returns a lot always produces a canonical lot, i.e. there are no duplicate keys.

One sample usage is in implementing enumerated types, whereby strings map bidirectionally to non-negative integers. Taking this a step further, a dense tabular data structure can be efficiently implemented as a list of rows, each being a list of cells, coupled with a lot mapping between column names and indices. The string representation would avoid the redundancy encountered when each row is a dictionary mapping from column names to cell values.

Proposed [lot] Subcommands

A functional programming style is more flexible and elegant than an imperative programming style; it frequently avoids the need for temporary variables which can clutter a program or collide with each other if not carefully named. Therefore most [lot] subcommands in Brush operate on values (as opposed to variables), return their results, and have no side effects.

The proposed functional commands are as follows:

Functional Commands Operating on Values	
`lot contains` lot key	True if lot contains key
`lot create` ?key ...?	Construct a lot given its keys
`lot difference` ?lot ...?	Symmetric difference of lots
`lot empty` ?lot ...?	True if all lots are empty
`lot equal` ?lot ...?	True if all lots are equal
`lot exclude` lot ?key ...?	Remove some keys from lot
`lot include` lot ?key ...?	Add some keys to lot
`lot index` lot index	Return index'th key in lot
`lot info` lot	Hash table statistics for lot
`lot intersect` ?lot ...?	Intersection of all lots
`lot intersect3` lot1 lot2	List: (lot1 – lot2, lot1 ∩ lot2, lot2 – lot1)
`lot search` lot key	Return index of key in lot, or `-1`
`lot size` lot	Number of keys in lot
`lot subset` lot1 lot2	True if lot1 is a subset of lot2
`lot subset -proper` lot1 lot2	True if lot1 is a proper subset of lot2
`lot subtract` lot1 ?lot ...?	lot1 sans keys in subsequent lots

Functional Commands Operating on Values	
`lot superset lot1 lot2`	True if `lot1` is a superset of `lot2`
`lot superset -proper lot1 lot2`	True if `lot1` is a proper superset of `lot2`
`lot union ?lot ...?`	Union of all lots

For the sake of convenience, a few imperative commands are proposed which operate on variables given their references:

Imperative Commands Operating on Variables	
`lot set lotref ?key ...?`	Add some keys to lot named by `lotref`
`lot unset lotref ?key ...?`	Remove some keys from lot named by `lotref`

The organization of the [`lot`] subcommands may inspire a refactoring of the [`dict`] command, but that is not yet defined.

Enhanced Syntax

Tcl's syntax is simple, but that does not mean it's always simple to use. There are several situations where the simplicity of the syntax actually discourages safe, correct programming and instead leads novice programmers into bad habits. In other cases, the simplicity of the syntax leads to seeming inconsistencies which can only be deciphered once the programmer has developed an in-depth understanding of the language and its commands.

Brush proposes to give the Tcl syntax a face lift. The changes described below are not merely syntax sugar; they are intended to encourage the programmer to follow best practices[11] by making the right thing also be the easy thing. Other changes make the language less surprising in light of expectations established by other programming languages, plus there are a few neat experimental ideas.

Bridging Substitution and Command Contexts

The interpreter's goal is not to perform substitutions, it's to execute commands, and it only performs substitutions in order to assemble command lines. However, there are situations where substitution alone provides all the functionality needed, so the programmer wants a pass-through command whose sole purpose is to return its argument.

One such case is Tcl's new [`lmap`] command[12] which works like [`foreach`] except its return value is a list of results from each iteration of the script body. If the output list elements are to be made using substitutions, math expressions, or lists, it is necessary to use [`subst`], [`expr`] or [`list`], respectively.

---

11 http://wiki.tcl.tk/best+practices List of some best practices in Tcl.

12 http://tip.tcl.tk/405 The [`lmap`] command is a collecting loop with the semantics of [`foreach`].

However, [subst] and [expr] are dangerous if their argument is not brace-quoted to prevent double substitution. Also, Brush renames [list] to [list create] which takes longer to type.

The Tcl interpreter has long known how to do variable and script substitution and concatenation without the aid of a command, and Brush adds math expression and list constructor substitution (described below). Now that all these tasks can be done directly by the interpreter, it makes sense to offer a straightforward way to glue interpreter-driven substitution to commands such as [lmap].

Brush meets this need by providing a command that simply returns its first argument. The command is simply called [:] (a single colon), or the "pass-through command".

In this example, "$(...)" performs math substitution, and "(...)" constructs lists.

```
lmap (x y) (1 2 3 4) {: $(x + y)}       # 3 7
lmap f (y reas) {: And$f}               # Andy Andreas
lmap f (y reas) l (G K) {: (And$f $l)} # {Andy G} {Andreas K}
```

For comparison, here is how the above is written in current Tcl:

```
lmap {x y} {1 2 3 4} {expr {$x + $y}}
lmap f {y reas} {subst {And$f}}
lmap f {y reas} l {G K} {list And$f $l}
```

The [:] pass-through command directly implements the K combinator[13], which returns its first argument and ignores its second, though scripts typically rely on side effects from computing the second argument. For example, here is postincrement. (The "&"s in front of variable names will be explained in the section on references.)

```
set &x 5                    # 5
set &y [: $x [incr &x]]     # 5
```

The return value of [:] is $x, which is substituted before Brush tries substituting [incr &x]. The return value of the latter is ignored, but that's fine since evaluating it had the desired side effect of incrementing the variable. In this example, "$x"'s final value is 6 and "$y" is 5.

For interactive use, [puts] and [:] have the same effect, though the mechanism is different. [puts] prints its argument to stdout and returns nothing, whereas [:] returns its argument to the shell which prints it to stdout. [:] is shorter to type and will be used in later examples.

Also to benefit interactive use, [:] returns empty string if given no arguments. This helps when running a command that will return a very large amount of data that would overwhelm the display. For example, typing the following line into a Brush shell will display nothing:

```
set &data [chan read [open hugefile]]; :
```

Math Expression Substitution

One of the secrets to Tcl's simplicity and flexibility is that it delegates nearly everything to commands. Like Lisp and Unix shell scripts, Tcl is command-oriented, not expression-

13 http://wiki.tcl.tk/k The surprisingly useful K combinator returns its first argument and discards its second.

oriented, and its operators (quoting, delimiting, substitution, expansion) exist to construct command arguments. Therefore, math is not a part of Tcl proper, but rather something that is done by the [`expr`], [`if`], [`while`], and [`for`] commands.

This is a good example of a little language[14] significantly extending the capabilities of the Tcl base language, which otherwise can only divide scripts into commands and words, performing substitutions along the way.

Yet, though this may be an interesting and consistent philosophical orientation, it has proven to be inconvenient for many practical programs. Math is needed very frequently, and in most programming languages math is "instantly" available: simply write the expression, and it's done. In Tcl, it takes an extra nine characters per math expression ("`[expr {...}]`"), plus a dollar sign for each variable usage.

One common shortcut is to omit the braces, saving two shifted keystrokes. In most cases this appears to give correct results, so programmers use it. However, it also creates serious performance and security problems. It hurts performance because there is no single `Tcl_Obj` in which to store the math expression internal representation, and it is insecure because double substitution[15] opens the door to injection attacks[16].

Brush improves the situation by making the safe, fast, correct programming style be the easiest to type. It introduces a shorthand for math expression substitution: "`$(...)`" is equivalent to "`[expr {...}]`", where "`...`" is any legal math expression. This reduces the per-expression overhead from nine characters to three.

The Brush interpreter performs no substitution on the text between the parentheses of "`$(...)`"; all it concerns itself with is locating the final close parenthesis. All math, including variable substitution, is performed by the math runtime, and there is no danger of double substitution.

This breaks compatibility with Tcl which interprets "`$(...)`" to be a directive to substitute an array element value, where the array variable name is empty string. (STOOOP[17] famously stores class and instance data in the empty-string array local to the object's namespace.) Consequently, Brush forbids using empty string as a variable name.

Brush retains the [`expr`] command for rare situations where the math expression (as opposed to any variables contained within) is legitimately dynamic, such as in a calculator application. To discourage abuse, automatic concatenation is removed; [`expr`] takes exactly one argument.

In addition to shortening the preferred syntax for performing math substitutions, in limited (but common) circumstances Brush relaxes the requirement to precede variable names with a "`$`" dollar sign.

---

14 http://wiki.tcl.tk/little+language In effect, each Tcl command defines its own domain-specific little language.

15 http://wiki.tcl.tk/double+substitution It is dangerous to reparse substitution results and do further substitution.

16 http://wiki.tcl.tk/injection+attack An injection attack is the substitution of executable code into an expression.

17 http://wiki.tcl.tk/stooop Simple Tcl-Only Object-Oriented Programming. Newer Tcl OOP systems exist.

If the variable name consists of only alphanumerics and underscores, and it does not start with a numeral, the dollar sign is optional. When using this shortcut, the variable must be named literally (i.e., no nested substitutions); it must not be named "`eq`", "`ne`", "`in`", or "`ni`"; and no indexing or dereferencing can be used. (See the section on substitution for details.)

These restrictions avoid ambiguity between variable substitutions, literals, function calls, array element names, and operators.

Any sequence of two or more "`:`"s within or adjoining a variable name is treated as a namespace separator, not a ternary operator case separator.

Here are some examples and side-by-side comparisons:

Tcl	Brush	Description
`expr {cos($x * 2)}`	`: $(cos(x * 2))`	Cosine of two times $x
`expr {cos($arr(x) * 2)}`	`: $(cos($arr(x) * 2))`	Literal array index/dict key
`expr {cos($arr($x) * 2)}`	`: $(cos($arr($x) * 2))`	Index/key comes from $x
`expr $formula`	`expr $formula`	Dynamic math expression

List Constructors

In Tcl, the preferred way to construct a list from non-constant elements is with the [`list`] command. Each argument to [`list`] is an element in the returned list. The [`list`] command is crucial to any well-written Tcl program, but it is clumsy and therefore is avoided by novices who discover that double quotes are easier to type and give deceptively similar effects. Double quotes actually behave like [`concat`] which concatenates its arguments, collapsing one level of nested list structures, and corrupting the results when the elements contain spaces.

Even though I have coded Tcl for over a decade, I still sometimes look for ways to avoid [`list`]. Recently I had some deeply nested list containing almost entirely static data, but some buried element was variadic. Rather than code the whole thing using [`list`] so I could use a normal substitution, I used braces then applied [`string map`] to inject the element I needed. This is certainly not the most efficient implementation, but it is more readable despite its complexity.

Brush introduces a shorthand for [`list`]: parentheses. Parentheses at the beginning of a word behave like a new quoting mechanism. They nest like braces, but interior word boundaries are respected and substitutions are performed. There is no need for backslashes at the end of every line. The result is a pure list[18].

This shorthand provides an interesting possibility: reclaiming the [`list`] command for use as an ensemble. The [`lindex`], [`lrange`], etc. commands become [`list index`], [`list range`], etc. subcommands, organized the same as [`string`], [`dict`], and others. (The full complement of new [`list`] subcommands is not defined at this time.) The drawback is longer command

18 http://wiki.tcl.tk/pure+list A pure list is a value with list internal representation and no string representation.

names for common operations, but Brush also offers streamlined notation for list and dictionary access, discussed later.

Parentheses have no special meaning inside double-quoted and braced words, only at the "top level" of interpretation and nested within other parenthesized words.

The "**{*}**" expansion operator can be used to selectively force individual list elements to be split into multiple elements on embedded word boundaries. When "**{*}**" is used with every element in a parenthesized list, the effect is the same as if [**concat**] were used instead.

I envision this new notation being used for nearly all list construction. Brace quoting will be relegated to nested code (e.g., [**proc**] bodies and regular expressions) and for the interpreter-generated canonical representation of strings and lists requiring quoting.

Brace-quoted lists conflict with variadic elements, and the [**list**] command is a chore to type and visually clutters the code. The beginner temptation is to surround the list with double quotes, but this breaks when the substituted elements contain whitespace, mismatched braces, etc. Parentheses provide an attractive and sensible alternative.

Brush adds parenthesized lists not only to the main interpreter, but also to the math expression syntax. Within the context of a math expression, a list is constructed by surrounding it with parentheses and separating the elements (which are themselves general math expressions) with commas. To resolve the ambiguity between a single-element list and a simple parenthesized expression, a single-element list has a comma immediately before the closing parenthesis. The "**{*}**" operator is supported.

```
set &var (\$var value)        # {$var} value
: (     a      b  c )        # a b c
: (a ( b c ) { d e } \{ ((
    )) " f g " $var)          # a {b c} { d e } \{ {{}} { f g } {$var value}
: ($var {*}$var)              # {$var value} {$var} value
: $("a" in ("a", "b", "c")}) # 1
: $(($var, (1, 2), (3,), (),
"x y", ("x y",), {*}$var))    # {$var value} {1 2} 3 {} {x y} {{x y}} {$var} value
```

Sexagesimal Notation

For my work in geographic information systems, I frequently use sexagesimal (base-60) notation to express latitude and longitude. To date, I have not found any languages with direct or library support, so I have to implement it myself. It occurs to me that it might be a useful feature to have in Brush, serving as an alternate way of expressing a floating-point value. Sexagesimal is useful not only for GIS but also for timekeeping, since minutes and seconds are base-60 for time as well as angles.

Brush sexagesimal values are two or three nonempty strings of decimal digits separated by apostrophes "'". The value may have an optional "**+**" or "**-**" sign prefix. An optional fraction suffix may be supplied, consisting of a period "**.**" and zero or more decimal digits.

The apostrophes divide the value into two or three fields, each of which is interpreted as decimal. The last field may have a fractional component. All fields except the first must be strictly less than 60.

When Brush encounters a sexagesimal value, it converts it to a real number by summing its fields. The second field is divided by 60, and the third field (if present) is divided by 3600. If the value has a "-" sign prefix, the sum is negated to get the final value.

Brush's [format] command gains a new "%D" conversion type which formats the value as sexagesimal. By default, or with the "h" (short) size modifier, the output has two fields (degrees and minutes). With the "l" (long) size modifier, the output has three fields (degrees, minutes, and seconds). The second and third fields are zero-padded to two digits. The precision specifies how many decimal places to give following the final field.

[scan] also gets "%D". With no size modifier, it autodetects the presence or absence of a seconds field. The optional "h" or "l" size modifiers explicitly specify the number of fields.

The mnemonic for "%D" is "degrees". It is chosen because it's adjacent to the sequence "%e", "%E", "%f", "%g", and "%G" which are Tcl's real number formats. Please do not confuse "%D" with "%d" which formats decimal integers.

Here are some examples demonstrating the various formats:

```
: $(1'02)          # 1.0333333333333334 %D   or %hD
: $(-5'02.300)     # -5.038333333333333 %.3D or %.3hD
: $(+10'02'03)     # 10.034166666666666 %+lD
: $(-89'02'03.45) # -89.03429166666666 %.2lD
```

Comments

Tcl comments are quite different than those found in other languages. Most of the time they resemble Python, Perl, and Unix shell comments, but there are some subtle discrepancies:

- Even though Tcl comments continue until the end of the line, a closing brace will end the line, even if it appears to be inside the comment. (Syntax highlighters usually handle this incorrectly.)
- A comment can only start where a command could start, so it cannot go on the same line as another command without an intervening semicolon, and it cannot be placed inside a list.

Brush comments are designed to more closely match user expectations for scripting languages.

Brush's line comments continue to the end of the line, even if there are closing braces. This requires a major change to the brace counting mechanism, described in the next section.

There are situations where it is desirable to have both a comment and a closing brace on the same line, so Brush adds block comments which are akin to C's "/*...*/" mechanism. The notation for Brush's block comments is "#{...}#". Block comments nest, so they can be used to comment out large sections of code, even if they already contain block comments.

Brush offers more flexible comment placement than Tcl. Line and block comments can start wherever *any* word of a command can begin, not just the *first* word, so it is no longer necessary to precede the pound sign with a semicolon when the comment is on the same line as the code it documents. Comments can be embedded in parenthesized lists, not just scripts.

Comments are only stripped from parenthesized lists, not from lists quoted with braces, backslashes, or double quotes. This ensures two things:

- Comments are handled only by the parser, not the string-to-list conversion function.
- Braces, double quotes, or backslashes can quote a pound sign in a parenthesized list.

Here is an example showing how Brush comments can fit inside the last argument to [**switch**], which is a list alternating between patterns and scripts[19]. In Tcl, the comments can only go inside the scripts, but Brush also lets them go *between* the scripts when the list is constructed using parentheses.

```
switch $value (
    # first check option-*
    option-1 {puts something #{print something}#}
    option-2 {puts #{print the value}# $value}
    #{ comment this out until it's debugged...
    option-3 {putz oops #{mysterious error?}#}}#
    # now handle everything else
    default {puts "don't know what to do!"}
)
```

Block comments behave a lot like quoted words. The opening sequence "**#{**" is only recognized if it appears at the very start of the word, and it is illegal for any word characters to follow the closing "**}#**". Of course, the major difference between a block comment and any other kind of word is that it produces no output. In that sense, block comments are like empty string preceded by "**{*}**".

Brace Counting

Tcl novices are frequently surprised by brace counting. The current Tcl behavior is very simple: count any brace that is not preceded by an odd number of backslashes. While this works in most cases, it clashes with the C-inspired user expectation that braces in double quotes and comments do not count. Brush has a more sophisticated brace counting scheme that skips braces in quotes and comments.

Here is some faulty Tcl code that looks like it should work fine:

```
proc test {x} {
    if {$x} {
        puts "{"
    } else {
        puts "}"
    }
}
```

19 Thanks to duck typing, such an alternating list can also be thought of and processed as a dictionary.

Calling [`test 0`] produces no output because there is actually *no* ***`else`*** *argument* to [`if`]. This is because the braces within double quotes effectively "quote" the `else`. Calling [`test 1`] prints an open brace, then fails with "`extra characters after close-brace`", referring to the close quote following the close brace.

Within Tcl, the fix is to precede the quoted braces with backslashes. However, the need for extra quoting goes against user expectations and is therefore a common source of errors.

In Brush, braces do not count toward the open/close count when they appear inside double quotes. This corrects the specific problem experienced by the above code. The brace counter maintains a state machine tracking how each character and word it encounters will be interpreted during execution[20]. For example, if the first character of a word is a quote: start quote mode, so the word extends to the matching quote; in the interval, do not count braces.

Backslash-Newline

In Tcl, there is one surprising instance in which brace quoting modifies the word: backslash-newline. Within braces, if a newline is preceded by an odd number of backslashes, the backslash-newline and any subsequent spaces and tabs are *replaced* with a single space.

To date, I have not been able to find any justification for this behavior. At the top level (outside of any braces) the Tcl interpreter already knows to treat backslash-newline as a word separator rather than a command separator.

Brush removes this oddity in order to simplify line number counting and to ensure the return value of [`info body`] matches the actual source code.

Formal Argument Lists

A Tcl formal argument list binds actual arguments to variables inside [`proc`] and lambda[21] scripts. In addition to simply giving a name for each argument, it can supply default values for omitted arguments and store excess arguments in a catchall variable. These features are useful but have some restrictions:

- The catchall argument must be named "`args`" and can only be the last argument.
- Arguments with default values cannot occur before normal, non-default arguments.
- It is very difficult to tell if an argument was omitted or explicitly set to its default.

Brush removes these restrictions. Formal arguments can be placed in any order, the catchall argument can have any name, and an argument can be optional yet have no default value.

The syntax for Brush formal argument lists is a little different than used by Tcl. Optional arguments have a question mark "`?`" appended to their name, and the catchall argument (which does not have to be called "`args`") has an asterisk "`*`" appended to its name. These extra characters are not part of the variable name; they are notation for the argument list.

---

20 http://wiki.tcl.tk/cloverfield+-+parser Partial implementation of a similar parser made for the Cloverfield project.

21 http://wiki.tcl.tk/apply Tcl lambdas are anonymous procs and are executed using the [`apply`] command.

Brush adds another style of argument not found in Tcl, called a bound argument. Its value is hard-coded into the formal argument list and is unaffected by the actual arguments. In fact, the caller will never know the bound arguments are even there. Bound arguments are specified in the same way as defaulted arguments, except equal sign "`=`" is used instead of "`?`". Bound arguments are useful when programmatically generating procs that need to capture some of their environment at creation time.

Optional formal arguments *may* (not must) be specified as two-element lists, the first element being the name (with trailing "`?`") and the second the default value. If the formal argument is instead a single-element list, and the caller omits the actual argument when calling the procedure, the variable is not created. The procedure can check if the variable exists to ascertain whether the actual argument was supplied or omitted.

It is an error for there to be fewer actual arguments than there are non-optional formal arguments. It is also an error for there to be more actual arguments than formal arguments when there is no catchall formal argument.

Brush binds actual arguments to variables by simultaneously iterating through the formal and actual argument lists, considering them in pairs. The iterators are not always in lockstep.

The algorithm is as follows:

- Non-optional arguments are directly assigned to local variables with the same name as the formal argument, advancing both iterators.
- Optional arguments are assigned only if the number of remaining actual arguments exceeds the number of remaining required formal arguments. If they are skipped, the formal iterator advances but the actual iterator stays put.
- Bound arguments are assigned using the values in the formal argument list. Just like defaulted arguments, the formal iterator advances and the actual iterator is untouched.
- The catchall formal argument collects however many remaining actual arguments are in excess of the number of remaining formal arguments. If the argument counts match or if at least one optional argument is omitted, the local variable is set empty.
- At the end of the actual argument list, if any formal argument remain, they must be optional or catchall. They are set to default values or empty list, respectively.

These rules ensure all required arguments are assigned, then allots extra actual arguments to optional formal arguments (giving preference to earlier arguments in the list), then finally gives whatever is left to the catchall argument.

Many Tcl core commands (e.g., [`lsearch`]) take options at the beginning of the argument list, rather than the end. Brush's expanded formal argument list specification makes this easy to implement, plus it allows for the catchall argument to be called "`options`" rather than "`args`"

if that makes more sense. Other commands take variadic arguments in the middle, for example [lset], which might choose to name its catchall argument "indices".

Not all Tcl core commands map nicely to the Brush model. For example, [puts] would be better served by assigning arguments right-to-left rather than left-to-right. In situations like these, the command can simply fall back on letting the catchall collect most arguments, then processing them however it wishes.

I conjecture that this new feature will improve performance since it reduces how much work the script must do to implement more complex argument schemes.

This table demonstrates the various types of formal arguments and how actual arguments are mapped to formal arguments:

Proc Definition	proc &p (a b? (c? xxx) d (e= yyy) f* g? h) {...}							
Proc Invocation	**Argument Value; "Ø" If Variable Unset**							
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**
p 1 2	wrong # args: should be "p a ?b? ?c? e ?f ...? ?g? h"							
p 1 2 3	1	Ø	xxx	2	yyy		Ø	3
p 1 2 3 4	1	2	xxx	3	yyy		Ø	4
p 1 2 3 4 5	1	2	3	4	yyy		Ø	5
p 1 2 3 4 5 6	1	2	3	4	yyy		5	6
p 1 2 3 4 5 6 7	1	2	3	4	yyy	5	6	7
p 1 2 3 4 5 6 7 8	1	2	3	4	yyy	5 6	7	8
P 1 2 3 4 5 6 7 8 9	1	2	3	4	yyy	5 6 7	8	9

Multiple-Variable [set]

The first argument to [set] will always be a reference, which is distinct from any multi-element list. This makes it possible for [set] to also accept a list of references as its first argument, in which case it assigns to multiple variables at the same time. If [set]'s first argument has length greater than one, its second argument is treated as a list of values to be assigned in the manner of [foreach].

```
set (&a &b) (1 2)          #
: ($a $b)                  # 1 2
```

Like Tcl's [lassign] command, Brush's multiple-variable [set] returns a list of extra values that were not assigned to variables. If all values were assigned, it returns an empty list.

```
set (&a &b) (1 2)          #
set (&a &b) (1 2 3)        # 3
set (&a &b) (1 2 3 4)      # 3 4
```

This is useful for shifting one or more elements from a list into variables, such as when processing command-line arguments.

```
set &args (1 2 3 4)
set &args [set (&a &b) $args]
: ($a $b $args)           # 1 2 {3 4}
```

To get this shift behavior for only one variable, use empty string as a dummy second variable:

```
set (&a ()) ()            # not enough arguments
set (&a ()) (1)           #
set (&a ()) (1 2)         # 2
set (&a ()) (1 2 3)       # 2 3
```

Unlike Tcl's [`lassign`], multiple-variable [`set`] throws an error when there are not enough values to assign to all variables.

Enhanced Index Notation

Classic Tcl string and list indexes are integers or "`end`" optionally followed by a negative integer[22]. TIP #176[23] adds support for "`end+`" followed by an integer and for two integers connected with "`+`" or "`-`". These new forms are intended to simplify basic arithmetic in situations where it would have been necessary to use "`[expr {...}]`".

Brush replaces this plethora of supported formats with the original three options, yet it meets the goal of TIP #176 with an alternate, more flexible approach: In indexes, integers are generalized to instead be arbitrary integer-valued expressions.

If "`end`" is used as a prefix, the expression must begin with "`+`" or "`-`", and the expression's value is added to the end index to get the normalized index.

Since it is an expression, the index must be brace-quoted if it contains substitutions or whitespace. However, Brush expressions do not always require variables to be preceded by "`$`", so braces can be omitted in many common situations.

```
set (&x &str) (2 abcdef)
string index $str 0       # a
string index $str end     # f
string index $str end-1   # e
string index $str x       # c
string index $str end-2*x # b
string index $str end-1+1 # f
```

Substitution

One major goal for Brush is to provide the ability to name not only variables, but also individual dictionary or list elements within a variable's value. This minimizes the need for

22 http://www.tcl.tk/man/tcl8.4/TclCmd/string.htm#M9 String and list indexes have the same format.

23 http://tip.tcl.tk/176 This TIP adds simple index arithmetic capabilities to `TclGetIntForIndex()`.

accessor commands. **$**-substitution is empowered to do the job directly, even for complex, nested, hybrid data structures.

Dictionary and List Substitutions

Brush borrows and extends the Tcl array notation, though it drops the underlying concept of a Tcl array, that being a collection of variables.

Dictionaries in Brush can be accessed using Tcl array notation, yet they otherwise work like Tcl dictionaries and are first-class objects. In addition to dictionary indexing, Brush offers list indexing, using braces instead of parentheses to surround the zero-based numerical index.

```
set &var (a b c d)
: $var             # a b c d
: $var(a)          # b
: $var{3}          # d
```

Indexing can be cascaded to navigate nested data structures, and the two styles of indexing can be mixed for hybrid data structures.

```
set &nums (en (zero one two) fr (zéro un deux))
: $nums            # en {zero one two} fr {zéro un deux}
: $nums(en)        # zero one two
: $nums(en){1}     # one
: $nums(fr){2}     # deux
```

Substitutions can be nested for indirection.

```
set &lang fr
: $nums($lang)     # zéro un deux
: $nums($lang){0} # zéro
```

Indexed substitutions can be nested arbitrarily.

```
set &prefs (color blue lang en style classic)
set &rev (cero 0 uno 1 dos 2)
: $nums($prefs(lang)){$rev(dos)}   # two
```

Consecutive uses of a single style of indexing can be expressed either with multiple applications of the basic index notation or by giving an index "path" as a list within a single pair of parentheses or braces.

```
set &matrix ((0 5) (-5 0))
: $matrix{1}{0}                    # -5
: $matrix{1 0}                     # -5
set &contacts (bob (phone 555-1235 email bob@heaven.af.mil))
: $contacts(bob)(phone)            # 555-0216
: $contacts(bob phone)             # 555-0216
```

If such an index path comes from a substitution, it must be preceded by the "**{*}**" expansion operator, or it will be interpreted as a single index. As when constructing command arguments or a parenthesized list, "**{*}**" is equivalent to explicitly substituting in each list element in sequence.

```
set &rowcol (1 0)
: $matrix{$rowcol{0} $rowcol{1}}   # -5
: $matrix{{*}$rowcol}              # -5
set &lookup (bob phone)
: $contacts($lookup{0} $lookup{1}) # 555-0216
: $contacts({*}$lookup)            # 555-0216
```

The text between list index braces and dictionary index parentheses is treated as a list. Care must be taken when performing dictionary indexing using a literally specified key which is a list with non-unit length or is not a well-formed list. Such keys must be quoted with double quotes, braces, backslashes, or parentheses. If the key is the result of substitution, there is no danger; it is a single index by default, unless "**{*}**" is used.

```
set &flatmatrix ((0 0) 0 (0 1) 5 (1 0) -5 (1 1) 0)
: $flatmatrix((1 0))               # -5
set &contacts ("andy g" (email andrew.m.goth@gmail.com) "andreas k" (email ...))
: $contacts("andy g" email)        # andrew.m.goth@gmail.com
```

When a Brush list index substitution goes out of bounds, an error is generated. This is in contrast to Tcl [**lindex**] which returns empty string. Like Tcl [**dict get**], dictionary index substitution produces an error when the requested keys are not found.

If a programmer wants to follow a variable substitution with a literal "**(**", "**{**", or "**@**", he or she must use a backslash "\" to prevent the interpreter from interpreting the metacharacter. ("**@**" will be discussed later.)

```
set &user andygoth
set &host facebook.com
: $user\@$host    # andygoth@facebook.com
```

One feature not provided is the ability for a single path value to contain both dictionary and list indices. I cannot see a practical reason to build this into the language. If this feature is desired, a script can implement its own traversal mechanism:

```
proc &index (value path*) {
    foreach (&type &subpath) $path {
        switch $type (
            dict {set &value $value({*}$subpath)}
            list {set &value $value{{*}$subpath}}
        )
    }
    return $value
}
set &value (
    a (b ((x _ ) (y *) (z @ w ?)))
    c (d ((x \$)) e ((z \# w !) (z ~ w `)))
)
index $value dict (a b) list (2) dict (z)   # @
index $value list (1 0)                     # b
index $value dict (c d) list (0) dict (x)   # $
index $value list (3 3 0) dict (w)          # !
```

List Range Substitutions

Brush has a special form of list indexing which yields a list range rather than a single element. In this form, a colon is used to separate the start and end indices.

Conceptually, the indices refer not to the elements but rather to the spaces between them. The first index gives the space before the indicated element, and the second index gives the space after. The returned list range subtends all the elements between the selected spaces.

If the second index comes before the first, the range is empty; it refers only to the space preceding the first element. If the second index is omitted (but there is still a colon), it defaults to "`-1`". Since the second index refers to the space after the indicated element, "`-1`" corresponds to the space before the first element of the list. Consequently, omitting the second index always results in an empty range.

In range substitution, indexes before or after the end of the list are clamped to the list length.

```
set &data (a b c d e f g h)
: $data{0:end}      # a b c d e f g h
: $data{0:0}        # a
: $data{-5:2}       # a b c
: $data{end:end}    # h
: $data{1:end-1}    # b c d e f g
: $data{end-4:4}    # d e
: $data{3:3}        # d
: $data{3:}         #
```

A list range substitution may not have any further indexes applied to it. This is because it does not refer to any one element of the variable; instead it constructs a new value.

Python has a similar feature, though it is called slices instead of ranges. Python end-relative indexing works differently than Tcl or Brush indexing.

Strided List Range Substitutions

The second index may be followed by another colon and a nonzero integer expression giving the stride. The default stride is "`1`", but this can be overridden to skip elements and/or reverse the list.

If the stride is negative, the before/after space convention is reversed, and the second index defaults to "`end+1`" if not explicitly specified. For negative stride, the first index denotes the space following the element, and the second index denotes the space preceding the element.

```
set &data (a b c d e f g h)
: $data{0:end:2}    # a c e g
: $data{1:end:2}    # b d f h
: $data{end:0:-1}   # h g f e d c b a
: $data{end:end:-1} # h
: $data{end::-1}    #
: $data{end:0:-2}   # h f d b
: $data{end-1:0:-2} # g e c a
```

A stride of "2" is useful for getting a list of all keys or values in a dictionary. "-1" stride provides an easy way to invert a dictionary such that its former values map back to its former keys.

Computed Variable Names

Tcl $-substitution only allows indirection inside array element names. The [set] command is required if the base variable name is computed, i.e. involves a substitution. Looked at another way, variable substitutions cannot be nested in Tcl.

Brush changes this by adding more variable name quoting styles. Tcl supports "$var" for literal names consisting of alphanumerics, underscores, and "::" namespace separators. Tcl also supports "${var}" for arbitrary literal names. Brush additionally supports "$"var"" wherein "var" can involve any kind of nested substitution.

A simple example would be one variable containing the name of another: "$"$x"". The value of variable "x" names the variable whose value is the overall result of the substitution. In Tcl, this can be done only by "[set $x]", short of [eval] and quoting hell[24].

"$"var"" notation nests. There is no ambiguity between the opening and closing double quotes since only the opening double quote has a leading dollar sign. I do not expect this to be needed often, but it exists for the sake of generality. For example, "$"$"xy"$z"" means:

1. Concatenate the values of variables "x" and "y" to get "xy".
2. Get the value of the variable named "xy", which is called "$"$x$y"".
3. Concatenate that value "$"$x$y"" with the value of variable "z" to get "$"$x$y"$z".
4. The result is the value of the variable named "$"$x$y"$z", called "$"$"xy"$z"".

Written in Tcl, this would be "[set [set xy]$z]"; swap "$"" for "[set " and ""]" for "]". This notation remains valid in Brush; it is just no longer required.

It is important to note that looking up variables with single-argument [set] precludes using the list and dictionary indexing described above. This is because element indexing is not a general-purpose operator operating on values, but rather is a directive to $-substitution operating on variables. (However, "$[set ...](index)" notation can be used; see below.)

Brush does *not* have a "$$var" notation, which presumably would be shorthand for "$"$var"". This is done to avoid ambiguity. Without the double quotes, it would not be clear whether any subsequent indexes apply to the nested variable substitution or the outer variable substitution. Would "$$foo(bar)" mean "$"$foo(bar)"" or "$"$foo"(bar)"?

```
set (&xyz &v1 &v2) (abc x z)
: $"${v1}y${v2}"    # abc
```

24 http://wiki.tcl.tk/quoting+hell Complicated quoting means you are overlooking an easier approach.

Functional Substitution

In many functional contexts, the value being indexed is not stored in a variable, but is returned by a command. [`dict`] and [`list`] can certainly be used to index such a value, but `$`-substitution is extended to make its compact notation directly usable even in the absence of a variable. The syntax is "`$[script]`", and the result of [`script`] is the value being indexed.

Functional substitution only makes sense in combination with indexing. "`$[script]`" with no indexing is equivalent to "`[script]`".

For example,

```
some_command                # a b c d e f g h
: $[some_command]           # a b c d e f g h
: $[some_command]{0:end:2}  # a c e g
: $[some_command](c)        # d
```

Be careful not to confuse "`$[script]`" with "`$"[script]"`", which is ordinary variable substitution where the variable name determined by script substitution.

Dereferencing

In addition to dictionary and list indexing, `$`-substitution supports one additional directive: the "`@`" dereference operator. The precise meaning of references will be discussed shortly.

In a `$`-substitution, "`@`"s can follow the variable name and may be freely mixed with dictionary and list indexes. "`@`" takes the value in the variable (or element thereof), treats it as a reference, and tries to obtain the referenced value. Further indexing and dereferencing can follow "`@`" if the referenced value is a dictionary, list, or reference.

This example shows how to get a variable's value, given a reference to that variable:

```
set &x data                 # data
set &ref &x                 # &123
: $ref@                     # data
```

Dereferencing can be used repeatedly and in combination with other methods of indexing:

```
set &x (a 1 b 2)            # a 1 b 2
set &y (a 10 b 20)          # a 10 b 20
set &rx &x                  # &123
set &ry &y                  # &124
set &rrx &rx                # &125
set &rry &ry                # &126
set &rlist (&rrx &rry)      # &125 &126
: $rlist{1}                 # &126
: $rlist{1}@                # &124
: $rlist{1}@@               # a 10 b 20
: $rlist{1}@@(b)            # 20
```

You may wish to revisit these examples after reading the section on references.

Substitution Syntax Comparison

The following table summarizes all the valid forms of substitution by comparing the Tcl and Brush notations side-by-side.

Substitution Type	Tcl	Brush
Simple name	`$simple_name`	`$simple_name`
Verbatim name	`${name_with_metachars}`	`${name_with_metachars}`
Computed name	`[set name_with_substitution]`	`$"name_with_substitution"`
Functional	`[script]`	`$[script]`
Single list index	`[lindex $name index]`	`$name{index}`
Multiple list index	`[lindex $name i1 i2 ...]`	`$name{i1 i2 ...}`
Pathed list index	`[lindex $name path]`	`$name{{*}path}`
List range	`[lrange $name first last]`	`$name{first:last}`
Empty list range	`[list]`	`$name{first:}`
Strided list range	Not Easily Available	`$name{first:last:stride}`
Empty strided list range	`[list]`	`$name{first::stride}`
Array index	`$simple_name(index)`	Not Available
Single dict index	`[dict get $name key]`	`$name(key)`
Explicit single dict index	`[dict get $name key]`	`$name((key))`
Multiple dict index	`[dict get $name k1 k2 ...]`	`$name(k1 k2 ...)`
Pathed dict index	`[dict get $name {*}path]`	`$name({*}path)`
Dereference	`[upvar 1 $name var; set var]`	`$name@`

Notes:

- "`simple_name`" is a string of one or more alphanumerics, underscores, or "`::`" namespace separators.
- "`name_with_metachars`" is a string consisting of any characters except closing brace.
- "`name_with_substitution`" is any sequence of characters on which the interpreter will perform variable, backslash, and script substitution to determine the variable name.
- "`$name`" is any valid `$`-substitution except for list ranging. This definition is recursive.
- Functional substitution is only useful in combination with indexing.
- Pathed and multiple list/dictionary indexing can be combined in a single operation.
- The Tcl dereference example is approximate. Many possible implementations exist.

References

Brush's powerful new `$`-substitution mechanism is only half of the equation. What good is reading a variable if it can't be written in the first place? To create or modify a variable or element, it is necessary to name it without taking its value, then to pass that name to [`set`].

In Tcl this is very simple: write the variable name, and it's done. Like everything else, a variable name is nothing more than a string.

I wanted to do the same in Brush, but I also wanted to be able to name elements in the same way as in `$`-substitution. Sadly, these goals conflict. Brush's variable and element names are not limited to simple literals but are an expression language which the interpreter does not always try to parse correctly. To correct this problem, Brush has a special reference syntax used for naming variables and elements.

Difficulties with Naming

Much like unbraced [`expr`], simply writing the variable name (no leading "`$`") has numerous problems when indexing is applied:

- *Syntax errors.* "`matrix{2 3}`" is actually parsed as two words since the interpreter splits the word on whitespace. Remember, a word is only brace-quoted if its *first character* is "`{`"[25].
- *Security holes.* "`contacts($name)`" can delete all your files if "`$name`" came from some untrustworthy source who maliciously set it to "`[exec rm -rf ~]`"[26].
- *Impaired performance.* With "`data{$index}`", there is no single `Tcl_Obj` in which it is possible to cache the parsed form of the variable name; string concatenation and parsing must be done every time.
- *Surprising results.* Put that first example in context: "`[set matrix{2 3}]`". This is actually legal Tcl. It means to set the value of variable "`matrix{2`" to "`3}`", which is surely not what was intended.

Tcl array variables are similarly afflicted, though to a much lesser degree. When preceded by "`$`" they work without issue; otherwise quoting is required when the key contains whitespace.

These issues arise due to the interpreter's lack of support for variable names when not performing substitution. Without "`$`", the interpreter treats a variable name like any other word, even though that means processing whitespace in a way that appears inconsistent with variable substitution.

25 http://www.tcl.tk/man/tcl8.6/TclCmd/Tcl.htm#M10 If the first character of a word is open brace ("{"), ...

26 http://xkcd.com/327/ Humorous but cautionary depiction of such an injection attack coming from an unlikely source. However, I disagree with the moral ("sanitize your inputs"), since it is a band-aid for a problem that can be fixed more efficiently and thoroughly by *never reparsing substitution results*.

What's more, the interpreter does its own substitution, as with any other word, then [`set`] reparses the substitution results, performing another round of substitution in the process. This opens up the same security hole experienced by [`expr`] when its arguments were already substituted by the interpreter.

Introducing References

The similarities with [`expr`] are instructive. Brush could adopt Tcl [`expr`]'s solution and strongly recommend that the user brace-quote any variable names. However, this would likely lead to another problem experienced by Tcl [`expr`]: Because the consequences are not immediately apparent, programmers forget to use braces. Worse, substitutions embedded within names will not always be performed at the right times or in the right stack frames.

Taking a cue from its own solution to the [`expr`] problem, Brush instead defines a streamlined notation for declaring that a word is a variable name. The interpreter knows how to handle variable substitutions, so it is more than adequately equipped to handle variable names. All that is needed is a hint from the programmer to enable variable name mode. The interpreter, seeing this hint, knows that the word is a *value* that names a variable or an element thereof. Such a value is called a reference.

Brush prefixes a word with an "`&`" ampersand to indicate that it is a reference. "`&`" is chosen for this purpose because C++ already associates that symbol with the term "reference." This behaves like a quoting mechanism in several ways:

- It tells the interpreter to apply parenthesis matching and other variable naming rules, rather than merely looking for the next whitespace, to identify the word boundary.
- Only a whole word can be a reference, and the first character of the word ("`&`" in this case) determines how that word is treated.
- The output word, a.k.a. the value or the string representation, is not necessarily identical to the literal text of the script.

`&`-references support indexing, same as for `$`-substitution. The notation is obtained simply by writing "`&`" instead of "`$`", though "`&`" only has special meaning as the word's first character.

Brush forbids the use of a "bare" string where a variable reference is expected. This is done for consistency and to avoid the problems described in the previous section. To convert a variable name (itself contained in a variable) to a reference, simply use "`&"$name"`". As mentioned in the section on substitution with computed variable names, the double quotes are required to avoid ambiguous cases.

Since a reference is a value, it can be returned; passed as an argument; or stored in a variable, list, or dictionary. References do not always have to be typed literally; they can be the product of substitution. Whereas Tcl requires [`upvar`] or [`uplevel`] to access the caller's variables, in Brush they can be reached simply by using a reference passed from the caller.

Unlike C++ references, Brush references must be created explicitly, like C pointers.

Value of a Reference

A reference's value is "`&`" followed by the referent variable's interpreter-wide unique ID. If the reference has any indexing, it follows the ID using the same notation used in the script, albeit with embedded substitutions already performed.

References are strings, but that is not all they are. Let's compare references to Tcl I/O channels to illustrate by analogy.

In Tcl, an I/O channel is a string, e.g. "`file5`", but it is also a key in a hash table mapping to the internal data structure that actually implements the channel. Likewise, references index into an interpreter-wide variable table, e.g. "`&123`" for variable #123.

Tcl I/O channels are created using the [`open`] command which does two things: create the internal structures, and generate a unique string which maps to said structures. Similarly, when Brush executes a command line containing an `&`-reference, it creates a variable and an associated reference value.

Not all references are given unique values. If multiple references within a single stack frame refer to the same-named variable, the interpreter gives them all the same value, assuming they have the same element indexing. Contrast with Tcl I/O channels, where opening one file multiple times yields distinct channels.

Early and Late Binding

At the moment a reference is created, all embedded substitutions are immediately performed. In this way, references capture a snapshot of the interpreter state. At the time the reference is created, it decides forever which variable or elements thereof are being referenced. This is early binding.

In this example, a reference to a variable's list element is created. Two methods are used to make the reference, but the same reference, with the same index, is obtained each time. "`&r1`" uses direct substitution, whereas "`&r2`" uses the "`@`" dereference operator on a reference to the index variable. Because of early binding, changing the index variable after creating the references does not affect them.

```
set (&x &i &j) ((a b c) 1 &i)
set &r1 &x{$i}   # &123(1)
set &r2 &x{$j@}  # &123(1)
: ($r1@ $r2@)    # b b
set &i 2
: ($r1@ $r2@)    # b b
```

A reference can contain late-bound indexes which are not decided until it is dereferenced. This is done by using the "`^`" late-binding dereference operator when constructing the reference. This operator causes a normal "`@`" dereference operator to be placed in the reference value, and the "`@`" will not be processed until the reference itself is dereferenced.

Modifying the previous example to use "^" instead of "@" or bare "$" causes $r1 and $r2 to contain embedded references with "@"s to be applied at dereference time. This defers indexing, so changing the value of $i does have an effect. Notice that $r1 and $r2 each contain references to not only the "x" variable but also the "i" variable.

```
set (&x &i) ((a b c) 1)
set &j &i         # &124
set &r1 &x{&i^}   # &123(&124@)
set &r2 &x{$j^}   # &123(&124@)
: ($r1@ $r2@)     # b b
set &i 2
: ($r1@ $r2@)     # c c
```

Late binding can only be applied to indexes, not to the variable name. This restriction is necessary because the referenced variable must be clearly identified in order for garbage collection to work. If an existing reference value could be coerced to reference any variable, no variables could ever be finalized.

The "^" late binding operator is only recognized when constructing a reference, and even then only inside list and dictionary indexing. In every other context, it has no special meaning.

Building References from References

Given a reference stored in a variable "$ref", the value of the referent variable (or element) is obtained by "$ref@". To make a new reference to an element of that result, the notation is "&$ref@" followed by the additional indexing operators, for example "&$ref@{$i}".

To understand this, start with "&name{$i}" and recognize that "name" can be computed. Now use "$ref@" in place of "name" to get "&$ref@{$i}". This works even if "$ref" already contains element indexing or dereference operators. Such a thing is called an additive reference.

You may recall that $-substitution does not support "$$name", only "$"$name"". Therefore this "&$ref@" notation is an exception to the rule that references are constructed by writing the substitution that would yield the desired element, only with "&" instead of "$" up front.

Another way of looking at it is that the entire substitution (everything but the leading "&") is performed, though not to get the value, but rather to locate the element to which the reference will point.

```
set &var (a (1 2 3) b (4 5 6))
set &ref1 &var          # &123
set &ref2 &var(b)       # &123(b)
set &ref3 &$ref1@(b)    # &123(b)
set &ref4 &$ref2@{1}    # &123(b){1}
: $ref3@                # 4 5 6
: $ref3@{1}             # 5
: $ref4@                # 5
```

Just like with normal references, the "^" late-binding dereference can be used in the index components of an additive reference. It will be replaced with an "@" dereference operator in the output reference value.

```
set &var (a (1 2 3) b (4 5 6))
set (&k &i) (b 1)
set &ref1 &var(&k^)     # &123(&124@)
set &ref2 &$ref1@{&i^} # &123(&124@){&125@}
: $ref2@                # 5
set (&k &i) (a 2)
: $ref2@                # 3
```

"`&$name`" is illegal because references point to variables or elements, not anonymous values, and "`$name`" yields a value. However, "`&"$name"`" is valid for creating a reference given a variable *name.*

The [`ref link`] command, described later, provides another way to create additive references.

The inverse operation, removing indexing from an existing reference, is not defined at this time. A [`ref`] command ensemble could facilitate reference examination and manipulation.

Three Stages of Reference Processing

Compilation. When a script containing `&`-references is submitted to the interpreter for compilation, the `&`-references are transformed into bytecodes that will produce a reference value upon execution. The bytecodes may use substitution and concatenation to determine the variable name and/or the indexing. Nesting and dereferencing may also be employed.

Execution. Given the bytecodes emitted by the compiler, the bytecode execution engine makes a reference value. The engine checks if the reference names a variable already present in the local stack frame. If not, a new variable is created in the global variable table, and its name and ID are put in the local stack frame. The reference value consists of the variable ID and any indexing instructions.

Dereferencing. When the reference value is given to a C function that needs to access the variable, it passes the reference `Tcl_Obj` to functions that read, modify, or unset the variable or an element of its value. This can be done immediately after the reference is created, or it can happen some time later, maybe even after the variable's stack frame has exited.

References and [`set`]

As shown by examples throughout this paper, `&`-references are used as the first argument to [`set`] in order to create variables or modify their values. Naturally, [`set`] supports not only simple references to variables, but also references to variable elements. In this way, [`set`] can be used in place of Tcl's [`lset`] or [`dict set`].

```
set &x (a 1 b 2)    # a 1 b 2
set &x(a) 0; : $x   # a 0 b 2
set &x(c) 4; : $x   # a 0 b 2 c 4
set &x{1} 1; : $x   # a 1 b 2 c 4
```

As "`&x(c)`" shows in the above example, references can be constructed to nonexistent elements. [`set`]'ing them creates the element. This mirrors how variables are created initially: the reference exists *before* the variable is made.

Creating list elements is limited by the requirement that the index numbering have no gaps. Only indices "`0`" through "`end+1`" can be assigned. When assigning to "`end+1`" (or the equivalent absolute index), the element is appended to the list, so [`lappend`] is not needed.

```
set &x{end+1} a; : $x        # a
set &x{end+1} b; : $x        # a b
set &x{2} c    ; : $x        # a b c
```

Assigning to list ranges works like [`lreplace`]: the indicated range is replaced with the new value, which is treated as a list of elements. [`linsert`]'s behavior is made possible by assigning to zero-width ranges; the empty range is "replaced" with the new list. Empty ranges are constructed when the second element comes before the first, which is the default when a colon is used with no second index.

```
set &x (a b c)
set &x{0:} _         ; : $x   # _ a b c
set &x{1:} (1 2)     ; : $x   # _ 1 2 a b c
set &x{3:4} ()       ; : $x   # _ 1 2 c
set &x{end:} ((x y)); : $x   # _ 1 2 {x y} c
set &x{end+1:} z     ; : $x   # _ 1 2 {x y} c z
```

When the list range has a negative stride, the inserted element order is reversed.

```
set &x{0::-1} (a b c) ; : $x # c b a
set &x{1:2:-1} (x y z); : $x # c z y x
set &x{end::-1} (1 2) ; : $x # c z y x 2 1
set &x{-1::-1} (3 4)  ; : $x # 4 3 c z y x 2 1
```

Be cautious of negative stride. An above example shows that assigning to "`&x{end:}`" puts elements *before* the current last element, which may seem surprising but is consistent with non-range list indexing. Negative stride reverses the before/after space convention, so "`&x{end::-1}`" references the space *after* the last element, and "`&x{-1::-1}`" references the space *before* the first element.

Assigning to a range with non-unit stride is tricky. The concept is that only the elements included in the range are replaced with new elements. For the sake of sanity, Brush requires the replacement list to be empty or to have the same element count as the range.

```
set &x (a 1 b 2)             # a 1 b 2
set &x{0:end:2} (A B) ; : $x # A 1 B 2
set &x{end:0:-2} (3 4); : $x # A 4 B 3
set &x{0:end:2} ()    ; : $x # 4 3
```

References and [unset]

In Tcl, variables are created by [`set`]. Brush is slightly different; variables are created by `&`-reference constructors, and [`set`] gives them their initial value.

[`unset`], likewise, works a little differently. Tcl's [`unset`] destroys the variable, and it is no longer accessible. Brush's [`unset`] removes the variable's value, as if [`set`] had never been called. [`unset`] does not remove the variable name from the local stack frame, so newly created references to the same-named variable will have the same value as existing references.

After being [`unset`], the variable can be given a value again by passing its reference to [`set`]. So long as it has extant references, the variable remains in the interpreter's variable table, even though it might not always have a value. All references continue to point to the same variable; [`unset`] does not break this link. In this way, an unset variable (or reference thereto) can be used as a "null" distinct from empty string.

[`unset`] works not only with references to variables, but also references to elements of variables. [`unset`]'ing an element means to remove it, so [`unset`] obsoletes [`dict unset`] and zero-element [`lreplace`].

Applying [`unset`] to a list index removes the element, causing higher-indexed elements to be shifted down one slot. [`unset`] on list ranges behaves similarly. For strided list ranges, the indicated elements are all removed as if they had been [`unset`] one-by-one.

```
set &x (a b c d e f g)       # a b c d e f g
unset &x{0}           ; : $x # b c d e f g
unset &x{end-1:end}   ; : $x # b c d e
unset &x{1:end:2}     ; : $x # b d
unset &x(b)           ; : $x #
unset &x              ; : $x # can't read "x": variable is unset
```

It's instructive to look at the error messages caused by unset variables.

```
: $z      # can't read "z": no such variable
: &z      # &123
: $z      # can't read "z": variable is unset
set &z    # can't dereference "&123": variable is unset
```

1. The first line attempts `$`-substitution on a never-before-seen variable, so the error says "no such variable."
2. Next, a reference is created, though its value is not saved anywhere. The variable's reference count is momentarily two, then it drops to one when the result is ignored. The remaining reference is from the local stack frame which now maps "`z`" to "`&123`".
3. The third line again tries to get the value, but this time the variable reference is found in the local stack frame. However, the substitution still fails because the variable has never been given a value.
4. Last, `$`-substitution is eschewed in favor of single-argument [`set`]. Like "`$`", [`set`] sees that the variable has no value. Unlike "`$`", the error message does not contain the variable name. This is because [`set`]'s argument is a reference value "`&123`", which does not embed a variable name.

Using [`ref link`] to Link Variables to References

The new [`ref link`] command links a referent variable or element into the local stack frame. The result appears to work like a local variable, but all accesses to it forward to the original variable. Similarly, any attempts to create references to this new linked variable actually end up creating references to the original variable.

[ref link] takes two arguments: the existing reference and the new variable *name.* It is important to stress that the second argument is a name, *not* a reference. This is not so unusual; recall that [proc]'s second argument is also a list of names, not references.

```
ref link &old new
: &old              # &123
: &new              # &123
ref link &old(x) elem
: &elem(y z)        # &123(x y z)
```

[ref link] takes the place of [upvar]. Unlike [upvar], [ref link] has no need of a stack frame level argument. This is because all variables are effectively global; it is their *names* that are only recognized locally.

Brush retains [upvar] for symmetry with [uplevel] which is still required.

If more than two arguments are passed to [ref link], they are used as additional reference/variable name pairs, in the same manner as [upvar].

If [ref link] is given the name of an existing variable, the variable name is retargeted to the new reference, decrementing the reference count of the old variable, likely triggering its finalization. If the reference argument is empty string, the variable name is simply removed from the local stack frame.

[ref link] may be part of a larger [ref] ensemble featuring commands to examine and manipulate references.

Comparison with Tcl

Despite not being formally recognized by the interpreter, Tcl has references too; they are simply variable names. Names are values that can be passed around and used in stack frames other than the one in which they were created. However, for them to be used elsewhere, the relative or absolute stack frame must be known, e.g. "this is my caller's variable" or "this is a global variable", and then [upvar] or [global] can link the variable into the local stack frame.

In addition to the requirement that the stack level be passed around out-of-band, the referenced variable cannot outlive its stack frame. If this is a problem, the variable must be created globally, perhaps in a namespace.

This leads to a new problem: the variable's lifetime becomes indefinite, and the script must be careful to finalize it when it is no longer conceptually reachable. Also, the script is responsible for generating unique names for the global variables.

Tcl "references" can only name variables, not elements. (Tcl arrays are collections of variables, not values.) An element reference scheme could be devised, but it would have to be implemented using custom accessor commands; basic $-substitution would be unavailable, short of elaborate variable traces.

Brush's references eliminate the need for [upvar] and [global] in the case of commands accepting variable references (names) as arguments. The identity of the originating stack frame is irrelevant because references are indexes into an interpreter-wide variable table.

Brush retains [upvar] and [global] unmodified, plus it adds a [ref link] command to link a local variable to any variable or element thereof given its reference.

Since Brush variables are kept in a global table, they can survive their stack frame for as long as they are reachable through references. This makes it easier to create anonymous mutable data objects accessible only to parts of the code which have been given the reference.

Brush references can name elements as well as entire variables, and indexed and unindexed references can be used interchangeably. This design largely obsoletes Tcl arrays, which are collections of variables that can be managed individually or as a group.

Brush retains Tcl's variable traces, though it loses out on array traces. An alternative to array traces may yet be defined, but it would face difficulties due to the element traces being on values rather than entire variables.

Reference Syntax Summary

The same notes apply to this table as for the substitution syntax summary.

Reference Type	**Syntax**
Simple name	`&simple_name`
Verbatim name	`&{name_with_metachars}`
Computed name	`&"name_with_substitution"`
Additive	`&$name@`
Single list index	`&name{index}`
Multiple list index	`&name{i1 i2 ...}`
Pathed list index	`&name{{*}path}`
List range	`&name{first:last}`
Empty list range	`&name{first:}`
Strided list range	`&name{first:last:stride}`
Empty strided list range	`&name{first::stride}`
Single dict index	`&name(key)`
Explicit single dict index	`&name((key))`
Multiple dict index	`&name(k1 k2 ...)`
Pathed dict index	`&name({*}path)`
Early-binding dereference	`&name{$name2@} or $name($name2@)`
Late-binding dereference	`&name{$name2^} or $name($name2^)`

Garbage Collection

Garbage collection in Brush is only preliminarily specified. It is a work in progress, and I enthusiastically invite suggestions to refine or replace the scheme defined here.

Tcl I/O channels must be explicitly destroyed using the [`chan close`] command. This is very different from Brush references (actually, variables), which are garbage-collected. When a Brush reference value's refcount[27] drops below one, it is finalized, and its referent variable's reference count is decremented, triggering cascading destruction whenever the value or variable does not have multiple referrers.

This simple scheme is defeated by circular references which prevent otherwise unreachable variables from ever being cleaned up. I am not well-versed in advanced garbage collection algorithms, but Frédéric Bonnet's Colibri[28] implements an exact, generational, copying, mark-and-sweep, garbage collector, so perhaps Colibri can be used to store values in Brush.

References and Shimmering

Reference tracking challenges EIAS semantics. If a reference shimmers away from its reference internal representation, despite keeping the same string representation, should the referent variable's refcount be decremented? Answering "yes" breaks EIAS, but handling this corner case will be costly.

One possible solution is to not decrement when the value is not actually changing, merely shimmering, but that leads to another question: when should the variable's refcount eventually be decremented?

In order to check if a value object contains references, it is necessary to *attempt* to convert it to a reference or list of references. Clearly, this is an expensive operation which should be done as infrequently as possible:

- This check is only done when the value is being destroyed.
- Type conversion is only attempted if the value is flagged as possibly being a reference.
- If the value's internal representation is a list, its contents are recursively scanned to find flagged values or nested lists. (Remember, a dictionary is a list.)

Value objects with internal type of reference, such as those generated by an `&`-reference constructor, have their reference flag set, as do any values made by concatenating a flagged value. Lists and dictionaries containing flagged values are *not* themselves flagged, at least not until they shimmer to pure string, at which point their values are made via concatenation with flagged values. Shimmering back to list or dictionary clears the flag, since it is transferred to any of the contained values which fit the reference schema.

27 http://www.tcl.tk/man/tcl8.6/TclLib/Object.htm Official documentation of `Tcl_Objs` and their refcounts.
http://wiki.tcl.tk/tcl_obj+refcount+howto Description of how to properly manage `Tcl_Obj` refcounts in C code.

28 http://wiki.tcl.tk/colibri Colibri is the string and data type infrastructure implemented by FB for Cloverfield.

Unshared string values can be modified in-place, possibly adding or removing a reference in the process. (This is not typical usage; the list and dictionary commands are preferred, and they have no need of flags since their elements are distinct values.) When C code modifies a string value in such a way that could add or remove a reference, it must call a function to check the modified string or substring for references and to update the flag accordingly.

Regarding "reference or list of references": This phrase interacts badly with EIAS. It collides with a limitation of duck typing, being that the interpreter only knows the type of a value when the script tells it the type. Here, the interpreter is forced to guess types. Its algorithm is as follows:

1. A value is a reference if it has reference internal type or can be converted to one.
2. Else, if a value has list (which includes dictionary) internal type or can be converted to a list with length *not equal to one*, it is a list. In this case, recursively apply this algorithm to each element.

Circular References

The scheme described above is vulnerable to circular references, e.g. [`set &x &x`]. Since its value refers to itself, the "`x`" variable's reference count will never drop below one and will never be destroyed. Flushing out circular references requires expensive reachability analysis.

When a stack frame is destroyed, the refcounts of all its variables are decremented. The frame's surviving variables are checked for reachability from any active stack frames, including those of all coroutines, within the current interpreter. This test is also done when a variable is removed from the stack frame using [`ref link`] but it still has positive refcount.

The reachability search is performed in an order designed to check the most likely places first:

1. Current stack frame, in the [`ref link`] case.
2. Late stack frame's returned value, in the [`return`] case.
3. Global stack frame, including all namespaces.
4. Current coroutine's stack frames, if in a coroutine.
5. Main routine stack frames.
6. All inactive coroutine stack frames.
7. Additional object stores registered by extensions with data "outside" of the interpreter.

Any variables found to be unreachable are destroyed. Reachability is defined as being referenced by a value object within, or reachable from, a stack frame. Only values flagged as possibly containing references are checked for references. The process of checking causes shimmering to reference or list.

The reachability search may be optimized by maintaining an interpreter-wide list of value objects contained within any of the searched stack frames.

Performance

The foregoing may sound extremely expensive, but it is made necessary by two corner cases:

- References embedded in strings that *could* be lists.
- Circular references.

I expect these cases to be rare in practice, so I designed this preliminary garbage collection algorithm to minimize cost in common cases.

If the script avoids shimmering away from reference or list-of-references, the only reference-flagged objects will in fact have internal type of reference, and the interpreter will never try to shimmer values back to reference or list-of-references.

[`proc`] bodies legitimately using circular references can significantly improve performance by breaking the reference prior to using [`ref link`] or [`return`].

Circular reference reachability analysis is only done when variables outlive their stack frames or survive [`ref link`]. This happens when a procedure returns a reference to a local variable, or if it puts a local reference inside a [`proc`] or other such object. The variable lives on, but after being separated from its stack frame, it can only be accessed via pre-made references.

In the course of shimmering a reference-flagged value to list, its reference flag is cleared, so the shimmering is a one-time thing. Basically the interpreter allows the script to shimmer from reference or list-of-references, but it forces the type back when the value is being destroyed or a reachability analysis is being done. Due to EIAS, this shimmering does *not* result in any values being changed, and the only cost of shimmering is CPU time.

Alternatives

As mentioned above, Colibri may provide some solutions, and it may be directly usable by Brush. I have not delved into its implementation to see precisely how it works.

Jim references[29] track values, not variables. Values persist so long as references to them exist, and references are values. Brush's design depends on references naming variables rather than values, so this is a fundamental difference.

Jim does not attempt to clean up circular references. Brush could adopt this as a design constraint, since Jim is successful despite this limitation.

Jim scans all existing value objects, including those not reachable from any stack frame, for the sake of simplicity and to avoid prematurely destroying references contained only "inside" an extension not fully exposed to the interpreter. Brush could do the same, at the expense of not being able to detect circular references.

Jim skips values that are not pure strings because they cannot be references. However, this assumption is invalidated by regular expressions and possibly other types. This limitation

29 http://wiki.tcl.tk/jim Jim is a small-footprint reimplementation of Tcl with some advanced features.
http://wiki.tcl.tk/jim+references Description, demonstration, and discussion of the Jim references system.

may be acceptable because there is no legitimate reason for a script to treat a value as both a reference and a regular expression.

Jim is documented to skip strings that are not exactly references, e.g. they have the wrong string length (Jim references are always 42 characters long, for performance and cosmic reasons.) This does not match the current implementation: Jim checks all pure strings to see if they contain substrings that are valid references. This is done to handle the case of reference lists shimmering to string. Brush probably can't do the same, since Brush references are harder to detect than Jim references.

Command Dispatch

To support its functional programming goal, Brush redefines command dispatch in a way that makes commands be values. A variable containing a command value is directly executable, or an anonymous command value can be invoked. Since command values are stored in variables, they enjoy the scoping and lifetime rules of variables, plus they can be passed around by value or reference as well as by name, or can be put inside data structures.

This design eliminates the value/command dichotomy[30] present since Tcl's inception. Tcl's [`apply`] command does this as well, but it must be used explicitly. Brush makes it automatic, plus it opens the door for more types of commands.

Unlike Tcl, Brush command and variable names can collide; they compete for the same "namespace". This is a drawback for some scripts that use a variable named the same as a proc to store the proc's static data[31]. However, as will be discussed later, it is possible to instead keep the data inside the proc itself, persisting from one invocation to the next.

$**-substitution is implied for the first word of each command. This has two consequences. One, a command name is actually the name of a variable containing the command. Two, the command name can use any of the indexing notations valid for **$-substitution. If the command is named via explicit **$**-substitution, the automatic **$**-substitution is inhibited.

When a command is invoked, the local stack frame is searched, just like ordinary **$**-substitution. If that search fails, the local stack's home namespace is searched, then the global namespace "**::**". As a last resort, the interpreter's [`unknown`] command is called.

The value of a command is a list. The first element of this list is the command type, and subsequent elements vary from type to type. The command type word is not the name of a command; it is handled internally by the bytecode compiler and/or execution engine. It is possible for a command to have the same name as a command type.

In addition to the string/list representation, command values have an internal representation containing executable bytecodes and/or type-specific configuration data. Beware that

30 http://wiki.tcl.tk/getting+rid+of+the+value/command+dichotomy+for+tcl+9 Commands are not first-class objects.

31 http://wiki.tcl.tk/gadgets Gadgets are objects with code in a proc and data in a variable, both named the same.

performing list commands (even ostensibly read-only list indexing, such as "`$cmd{$i}`") will cause the command value to shimmer away from its compiled command representation. This does not impact program correctness but does force a time-consuming recompilation. Nevertheless, it may be useful within debug contexts or for very dynamic coding techniques.

Each command type can have an associated finalizer routine which cleans up any exterior resources and data structures associated with the command.

One major difficulty for introspection and error message generation is that command values are anonymous. They only borrow the name of their container variable, but that does not help in situations where the command value is substituted and/or computed on the spot.

Lambda Commands

A lambda[32] is an anonymous proc. Its value is a list with three or four elements:

1. The word "`lambda`". This distinguishes between lambdas and other command types.
2. Formal argument list, which can include required, optional, defaulted, catchall, and bound arguments.
3. Script body. This will be executed in a new stack frame initially containing one variable for each formal argument.
4. Namespace *(optional)*. When the script invokes a command or calls [`variable`] to link a variable into the local stack frame, this namespace is searched before looking in the global "`::`" namespace. By default, the namespace is computed from the command name (if the command is global) or inherited from the local stack frame (if local).

A Tcl-like [`proc`] command can be implemented simply:

```
set &::proc (lambda (nameref arglist body) {
    set $nameref (lambda $arglist $body); :
})
```

This creates a lambda and binds it to the name [`proc`] in the global namespace "`::`". When [`proc`] is later executed, it constructs a lambda from its arguments and binds it to the variable indicated by its first argument.

Bound arguments can be used to capture the procedure's creation-time environment. If the bound arguments are set to references[33], and the same references are given to other procs, the procs will be able to share some variables, thereby establishing an object system. The references can be to variables local to the stack frame that created the procs, so they will be anonymous and inaccessible outside of the constructor procedure, and they will be finalized when the procs are destroyed.

32 http://wiki.tcl.tk/lambda Lambdas are anonymous functions, or procs in the Tcl/Brush parlance.

33 http://wiki.tcl.tk/closures Brush implements closures by allowing the programmer to bind arguments to references to variables local to the stack frame in which the proc is being created.

Reference-bound arguments can be turned into local variables using the [`ref link`] command to replace the argument variable with the referent variable. This avoids the constant need for the "`@`" dereference operator.

To demonstrate, here is a solution for Paul Graham's accumulator generator problem[34]. For the sake of example, this implementation deviates from Paul Graham's rules by offering defaults for `$value` and `$increment`.

```
proc &accum_gen ((val? 0)) {
    : (lambda ((valref= &val) (inc? 0)) {
        set $valref $($valref@ + inc)
    })
}
```

[`accum_gen`] takes an initial value argument from which it constructs a lambda that returns the sum of the initial value and its `$inc` argument. A reference to the initial value is bound to the lambda's `$valref` argument, so the lambda has sole access to a variable whose lifetime matches that of the lambda itself. The lambda, when executed, gets the value stored in that variable, adds `$inc` to it, stores the result into the variable, and returns said result.

```
set &accums (a [accum_gen 12] b [accum_gen 4])
accums(a) 0         # 12
accums(a) 5         # 17
accums(a) -2.5      # 14.5
accums(b) 6         # 10
```

Native Commands

Obviously, not all commands can be implemented in script; there must be a basis implemented in C. Native commands are commands written in C, etc. and compiled to machine code. Their implementation is opaque to the script, instead replaced by a numeric identifier. The value of a native command is a two- or three-element list:

1. The first word, "`native`", is the command type.
2. Numeric identifier for the command. This is *not* a pointer to the function; it is an index into the interpreter's native command table. This is done to prevent safe interpreters from calling unauthorized commands.
3. Namespace *(optional)*. This is only necessary if the command accesses namespace variables, which is vanishingly rare for native commands. If omitted, the namespace is determined in the same way as for lambdas.

In this example, [`set`] and [`:`] are revealed to be the 7th and 9th commands:

```
: $set              # native 7
: ${:}              # native 9
```

34 http://wiki.tcl.tk/accumulator+generator The challenge is to make a function that returns a function which returns the sum of all values ever passed to it. The accumulator's initial value is specified when the function is generated.

Curried Commands

Brush allows any command to be curried[35], meaning that one or more initial arguments are bound in advance. For example, a command that adds two numbers could be curried to make the first number always be "`1`", resulting in a command that increments a number.

A curried command's value is a list with at least two elements:

1. The word "`curry`" gives the command type tag.
2. Value of the command being curried, e.g. "`native 74`" or "`lambda {x y} {: $(x+y)}`". Since curried commands are commands, they can be nested.
3. Subsequent arguments serve as initial arguments to the command. A more efficient way to further curry a curried command is to append argument elements to its value.

The value of the above increment example is "`curry {lambda {x y} {: $(x+y)}} 1`", and it is constructed in a very simple and natural way:

```
proc &sum (x y) {: $(x+y)}
set &inc (curry $sum 1)     # curry {lambda {x y} {: $(x+y)}} 1
inc 5                       # 6
```

Prefix Commands

A prefix command is perhaps better termed a "command prefix[36] command"; it is a command value containing a command prefix. Since command prefixes are themselves commands, maybe the name could be "command command", except that's too confusing.

Prefix commands are almost identical to curry commands, except that the second list element is a command name instead of a command value.

```
proc &sum (x y) {: $(x+y)}
set &inc (prefix sum 1)     # prefix sum 1
inc 5                       # 6
```

Prefix commands are useful in situations where a command value is expected but you have only a command name. In that sense, it is the complement of the [`apply`] command, which enables execution of a lambda (which Brush generalizes to a command value) in contexts where a command prefix is expected.

Because the indicated command can be modified after the prefix command value is created, late binding can be implemented using prefix commands

Channel Commands

Tcl's I/O channels are represented in the interpreter by strings such as "`stdout`", "`file5`", or "`sock304`". They are constructed using [`open`], [`socket`], [`chan create`], [`chan pipe`], or extension commands, and they must be explicitly finalized using the [`chan close`] command.

35 http://wiki.tcl.tk/curry Curried functions accept arguments one at a time using nested single-argument functions.

36 http://wiki.tcl.tk/command+prefix A command prefix is a list containing a command name and some or all of its arguments, with the expectation that zero or more arguments will be appended and the result will be evaluated.

Brush I/O channels are constructed in the same way but are channel-type command values. This gives them several very interesting properties:

- Brush I/O channels function as command ensembles, so the [chan] command is unnecessary except for [chan names], [chan create], and [chan pipe].
- Values, including command values, are garbage collected. The finalizer for channel command values closes the channel. This makes explicit [chan close] optional, and [unset] can be used in its place. Additionally, garbage collection minimizes the need for [chan names].

The value of a channel is a list, and it is exceptionally simple:

1. The word "chan".
2. Name of the channel, same as in Tcl. For example, "stdout", "file5", or "sock304".

Global variables called $stdin, $stdout, and $stderr are pre-created, containing the values "chan stdin", "chan stdout", and "chan stderr", respectively. These variables work like commands (or objects, if you prefer), and they have subcommands à la [chan].

Here are some examples demonstrating a few usage possibilities:

```
stdout puts >>[stdin gets]<<    # copy from stdin to stdout, adding >> and <<
set &data [[open file] read]    # file is automatically closed after the read
set &out $stdout                # let stdout be accessed throughh another name
unset &out &stdout              # now both must be unset to close stdout
```

Interpreter, Coroutine, and Namespace Commands

Interpreters, coroutines, and namespaces are given the same treatment as channels; Brush promotes them all to be command values.

Ensemble Commands

Brush splits ensembles from namespaces because they can now be implemented directly inside a single value. Aside from the invocation, there is not much difference between an ensemble and a dictionary of command values,.

For demonstration purposes, here is a dictionary of command values. In this example, the keys are two-element lists, which is why the indexing is done using double parentheses: The inner pair is used to construct a list to be used as the key.

```
set &cmds ((msg 1) (lambda () (: hello))
           (msg 2) (lambda () (: goodbye)))
cmds((msg 1))                   # hello
cmds((msg 2))                   # goodbye
```

An ensemble command's value contains such a dictionary of command values, but it adds some configuration options. It is formatted as a two- or three-element list:

1. The word "ensemble".
2. Command dictionary mapping from subcommand names to command values. The keys are treated as lists to implement multiple-word subcommand names. It is an error for one key to equal or be a prefix of another key.
3. Configuration dictionary *(optional)*. If omitted, it is treated as if it were empty.

The configuration dictionary supports a subset of Tcl's [`namespace ensemble`] options[37]:

- "`parameters`": If present, the length of this list is the number of actual arguments accepted *between* the ensemble name and the subcommand name. Normally all arguments are expected to *follow* the subcommand name. The list elements are used as formal argument names to be displayed in error messages.
- "`prefixes`": If omitted or logically false, subcommand names must exactly match the keys in the command dictionary. If logically true, subcommand names can be any unambiguous prefix of the command dictionary keys.
- "`unknown`": If present and non-empty, contains a command value to be invoked whenever an unrecognized subcommand is called. The arguments to this command are the fully-qualified ensemble command name (if known, else empty string) and all its arguments, including the subcommand name(s). A prefix command may be useful here to use an existing command as the unknown handler, or the unknown handler can be specified inline and remain anonymous.

A command named [`ensemble`] may be defined to facilitate and optimize creating, querying, and reconfiguring ensemble command values. While ensembles can be managed using list and dictionary access, they have the side effect of shimmering away the ensemble command value's internal compiled representation.

Here is the previous example rewritten to use an ensemble command:

```
set &lookup (ensemble ((msg 1) (lambda () (: hello))
                       (msg 2) (lambda () (: goodbye))))
lookup msg 1       # hello
lookup msg 2       # goodbye
```

Object Commands

I have minimal experience with TclOO[38], but it seems likely an OO system can be built on top of Brush's command value mechanism. In the section on lambda commands, this paper already outlined ways to exploit references to share data between procs, such that the procs

37 http://www.tcl.tk/man/tcl8.6/TclCmd/namespace.htm#M34 Official documentation for ensemble options.

38 http://wiki.tcl.tk/tcloo TclOO is the first dedicated object-oriented system included in the official Tcl distribution.

together form an object with hidden data. The ensemble command system can further be used to neatly group those procs within a single object command value.

Object systems typically do more than just group code and data. They may offer inheritance, delegation, standardized interfaces, polymorphism, run-time type identification, and programmable finalization. To that end, Brush proposes (but does not yet specify) to adapt TclOO or a similar OO mechanism to an object command value.

Being able to customize finalization is the main reason why command values need to be a distinct command type. As already shown for channel commands, at the C level the command type registration system provides hooks for supplying a type-specific finalization routine. An object system would expose that hook at the script level, making it possible to script object cleanup.

Extension and Reflected Commands

Extensions can add custom command types. For example, Brush/Tk[39] may define window, image, and font command value types.

It may be useful to reflect this extension capability back into the script level so that new command types can be defined programmatically. In this way, fully custom object systems can be defined.

Conclusion

Brush offers an exciting palette of functionality designed to encourage programmer creativity and expression to blossom, as well as to attract new attention to the Tcl universe.

Everything in Brush is founded on the simple-yet-powerful Tcl **EIAS** philosophy. Within the framework of EIAS, Brush defines new reference and command values and integrates them with the interpreter, plus it unifies dictionaries and lists to optimize interchangeability.

Building upon Tcl, Brush streamlines and enhances the syntax to **encourage best practices** proven by long experience to promote efficiency and safety. The syntax improvements also make Brush more familiar to users of other programming languages.

References are exploited to establish a potent and compact **data structure access** notation through which even deeply nested, mixed data structures are easy to manipulate with minimal need for accessor commands.

Elegant **functional programming paradigms** arise from Brush's reference and command value design. References provide excellent control over variable access and lifetime, and command values are first-class citizens within the interpreter, fully exposed to the same powerful data manipulation infrastructure as any other kind of value.

---

39 http://wiki.tcl.tk/tk Tk is the most popular GUI toolkit used with Tcl.
http://wiki.tcl.tk/gnocl Gnocl is an alternative GUI toolkit for Tcl that binds to Gtk+.