Brush: A New Tcl-like Language

Presented by Andy Goth

19th Annual Tcl/Tk Conference

November 2012

Chicago, IL

History of Brush

- My Wibble web server uses deeply nested lists and dictionaries
  - Powerful data design
  - Clumsy to access
- Frédéric Bonnet proposed the Cloverfield project to investigate radical designs for Tcl 9
  - Some of Cloverfield's ideas would benefit Wibble
  - Common goals, but divergent approaches

Design Goals

- Brush has four primary design goals
  - Everything is a string
  - Streamline best practices
  - Enhance data structure access
  - Facilitate functional programming
- Tcl compatibility
  - Break syntax-level compatibility when necessary
  - Respect the Tcl design philosophies

Everything is a String

- Tcl’s great strength is its EIAS philosophy
  - Trivial serialization
  - Maximal compatibility
  - Easy introspection
- Brush embraces EIAS
  - EIAS guides the design of Brush’s new features

Dict/List Unification

- Brush's dicts are lists with hash table indexes
  - Can freely read dicts using list methods
  - Don't have to worry about shimmering
  - Hash table is automatically created, updated, and removed according to the way data is accessed
- New [`lot`] command for sets
  - No dummy elements in value
  - Constant-time index lookup given key

[lot] Examples

-
  ```
  lot contains   (a b c) a         # 1
  lot difference (a b) (b c)       # a c
  lot equal      (a b c) (b a c)   # 1
  lot exclude    (a b c) b c       # a
  lot intersect  (a b c) (b c d)   # b c
  lot search     (a b c d) c       # 2
  lot size       (a a b a c)       # 3
  lot superset   (a b c) (b c)     # 1
  lot union      (a b) (b c)       # a b c
  ```

-
  ```
  set &x (a b c)
  lot set &x d                     # a b c d
  lot unset &x b c                 # a d
  ```

Enhanced Syntax

- Tcl's simple syntax isn't always simple to use
  - [`expr`] unsafe and slow without brace quoting
  - [`list`] inconvenient for complex tree structures
  - Comments and braces can be surprising
  - Many [`proc`]s need to parse `$args`
- Brush builds on Tcl's syntax
  - Make the <u>right thing</u> be the <u>easy thing</u>
  - Be more accessible to new programmers

[:] Pass-Through Command

- In places where a command is expected, often only need substitution
- Pass-through command [:] simply returns its first argument
- Used in examples throughout this presentation
-
  ```
  : x                          # x
  : $var                       # value of var
  : a b c                      # a
  :                            #
  lmap f (y reas) {: And$f}    # Andy Andreas
  ```

“`$(...)`” Math Substitution

- [`expr`] unsafe and slow if argument not braced
  - Injection attacks
  - No bytecoding
  - Common mistake
- Brush adds “`$(...)`” notation, equivalent to but easier to type than “`[expr {...}]`”
- “`$`” before variables optional for simple cases
- `$(cos(x * 2))`

“(...)” List Constructors

- `[list]` is clumsy but essential
  - New or lazy programmers use double quotes instead
- Brush adds parentheses as a new quoting style
  - “`(...)`” equivalent to “`[list ...]`”
  - Similar rules as double quotes and braces
  - Substitution, nesting, comments, line breaks, “`{*}`”
- Also adds parentheses to expression notation
-
  ```
  : (a ( b c ) $var)        # a {b c} {x y z}
  : $((1, (("b c",), 2)))   # 1 {{{b c}} 2}
  ```

Comments

- Brush comments can start at any word
  - No need for semicolons
  - Can be used inside “`(...)`” lists
- Extend to line end even through closing braces
- “`#{...}#`” block comments support nesting
-
```
switch $value (
    # first check option-*
    option-1 {puts #{value}# >>$value<<}
    #{ commented out... }}}}{}{}}{
    option-2 {putz oops #{bug}#}}#
)
```

Brace Counting

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

- Above code broken in Tcl, works in Brush
  - Braces ignored inside double quotes or comments
  - Brace counter maintains state machine to figure out how characters will be interpreted at execution

Formal Argument Lists

- Brush enhances formal argument list notation
  - Reduce workload for common argument schemes
  - Increase flexibility
  - Support bound arguments
- `proc &p (a b? (c? xxx) d (e= yyy) f* g? h)`
  - High priority: required arguments
  - Medium priority: “`?`” optional arguments
  - Low priority: “`*`” catchall argument
  - Assigned in advance: “`=`” bound arguments

Other Features

- Sexagecimal (base-60) notation: “-89'02'03.45”
  - Alternative way to express floating-point numbers
- Backslash-newline inside braces
  - Tcl replaces with single space
  - Brush leaves unmodified
- Expression indexes
  - Instead of integer literals, allow integer expressions
- Multiple-variable [set]

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

Substitution

- New forms of substitution minimize need for accessor commands

Computed Name	`$"name_with_substitution"`
List Index	`$name{index}`
List Range	`$name{first:last}`
Dictionary Index	`$name(key)`
Dereference	`$name@`
Combination	`$name{idx1 idx2}@(key)`
Functional	`$[command]{index}`

References

- Brush adds variable references
  - References point to variables, not values
  - Can include indexing, same as substitution
- Variables are garbage collected
  - Circular references supported but expensive
- References are constructed using "`&name`"
  - Works like `$`-substitution with "`&`" instead of "`$`"
  - References are values

References and [set]

- [set] now takes a reference instead of a name
  - References can be passed around freely without regard for what stack frame they were created in
  - [set] can now access dictionary and list elements
  -
    ```
    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
    set &x{end+1:} (d 5); : $x # a 1 b 2 c 4 d 5
    set &x{end+1} x      ; : $x # a 1 b 2 c 4 d 5 x
    ```

Command Dispatch

- Brush commands are list values
  - First word is command type
    - lambda, native, curry, prefix, chan, ensemble
    - interp, coroutine, namespace, object
  - Remaining words vary by command type
- Command value comes from variable with same name as command
  - “**$**” implied at beginning of every command
  - Can use advanced substitution syntax with or without the leading “**$**”

Command Examples

- [proc] can be implemented using [set]

  -
    ```
    set &::proc (lambda (nameref arglist body) {
        set $nameref (lambda $arglist $body); :
    })
    ```
- Paul Graham's accumulator generator in Brush

  -
    ```
    proc &accum_gen ((val? 0)) {
        : (lambda ((valref= &val) (inc? 0)) {
            set $valref $($valref@ + inc)
        })
    }
    set &accum [accum_gen 12]
    accum 5        # 17
    accum -2.5     # 14.5
    ```

More Command Examples

- Currying is particularly easy in Brush

  -
    ```
    proc &sum (x y) {: $(x + y)}
    set &inc (curry $sum 1)
    : $inc  # curry {lambda {x y} {: $(x + y)}} 1
    inc 5   # 6
    ```
- Channels close automatically
  - When refcount drops below one, channel command finalizer routine is invoked
  - `set &data [[open file] read]; :`

Bringing It All Together

- A generator proc can return a command value or a list or dictionary of command values
  - Commands are first-class objects
- The command values can be lambdas with some arguments bound to references to variables local to the generator proc
  - Variables persist as long as references to them exist
  - Multiple procs can be given the same reference
  - This establishes closures and an object system

Summary

- Brush defines more flexible substitutions to improve data structure access
- Brush defines references to make writing variable elements work the same as reading
- Brush defines garbage collection to make references be more generally useful
- Brush redefines commands to be values
- Putting references to anonymous, GC'ed variables into command values opens wide the door to functional programming