Into the Interpreter

Into the Interpreter with ConsTcl

Peter Lewerin

Förlag: BoD · Books on Demand, Östermalmstorg 1, 114 42 Stockholm, Sverige, bod@bod.se
Tryck: Libri Plureos GmbH, Friedensallee 273, 22763 Hamburg, Tyskland

ISBN: 978-91-8097-023-5

Short contents

Contents

Introduction

To run the software

First things first. To run the software, source the file **constcl.tcl** (with **schemebase.scm** in the directory) in a Tcl console (I use **tkcon**) and use the command **repl** for a primitive command dialog. Source **all.tcl** to run the test suite (you need **constcl.test** for that). The files can be found on GitHub/ConsTcl[1].

The following license holds for the software, not the book:

MIT License

[1] See `https://github.com/hoodiecrow/ConsTcl`

modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The software is pretty much harmless, but there is a potential danger if anyone were to use it to write and run Scheme code that uses the output functions and in the process were careless enough to let the code overwrite existing files. I have put safeguards in the output functions that need to be removed before they work.

The software works for me, but there might still be problems that I haven't noticed. Contact me at

`constcl1000@gmail.com`

with any queries or error reports. I intend to keep updating the software to include bug fixes and interesting additions, within reason.

Background

It all started with Peter Norvig's Lisp emulator Lispy[2]. In January 2025 I was inspired to port it to Tcl. The result was Thtcl[3]. It had the same features and limitations as Lispy, plus a couple of limitations that were due to shortcomings of Tcl, and I came out of the experience feeling a bit dissatisfied. In the latter part of January I embarked on a new project, ConsTcl, a true Lisp interpreter. In Tcl.

About Lisp

ConsTcl is a *Lisp interpreter*, specifically a *Scheme interpreter*. Lisp is a family of programming languages that share the same basic form, and Scheme is one of those languages. Where some other languages use braces to structure code, and Python uses indents, Lisp instead uses parentheses. This Python snippet:

```
x = 1
if x == 1:
    print("x is 1.")
```

looks like this in Scheme:

```
(let ((x 1))
  (if (= x 1)
    (write "x is 1.")))
```

[2]See `https://norvig.com/lispy.html`
[3]See `https://github.com/hoodiecrow/thtcl`

In Lisp, everything is either an *atom*, an indivisible value, like `x` and `1` (and e.g. `let` and `write` too) or else it is a *list expression*, starting and ending with parentheses and having an operator at the front of the list, with the rest of the parts being operands. If the operator is a keyword like `let` or `if`, then the expression is a *special form*. If not, like `=` or `write`, then it's a *function call*.

A full description of Lisp is beyond the scope of this introduction, but Lisp will be explained from the inside out in the rest of this book. If you want to learn Scheme right away, there are good tutorials in different[4] places[5] on the web[6].

About ConsTcl

Compared to Lispy/Thtcl, ConsTcl has, (quote from Lispy), "comments, quote and quasiquote notation, # literals, the derived expression types (such as cond, derived from if, or let, derived from lambda), and dotted list notation." Again compared to Lispy/Thtcl, ConsTcl has the data types, quote, "strings, characters, booleans, ports, vectors." And pairs and procedures too. The number of missing primitive procedures is in the tens, not the 100s.

The completeness comes with a price: due to the sheer number of calls for each action, ConsTcl is is fairly slow. On my

[4] See `https://docs.scheme.org/schintro/`

[5] See `https://www.scheme.com/tspl4/`

[6] See `https://files.spritely.institute/papers/scheme-primer.html`

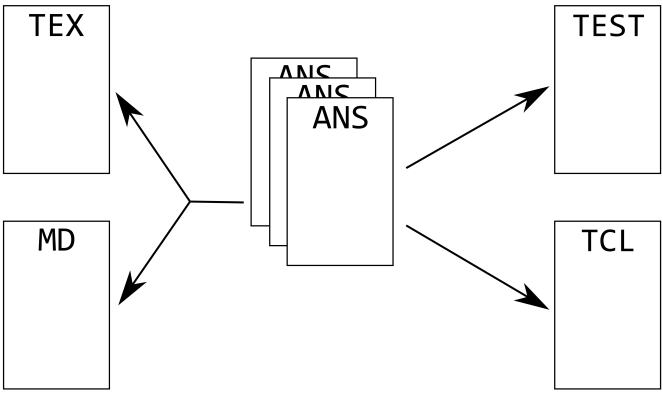
cheap computer, the following code (which calculates the factorial of 100) takes 0.04 seconds to run. That is thirteen times slower than Lispy assuming that Norvig's computer is as slow as mine, which is unlikely.

```
time {pe "(fact 100)"} 10
```

ConsTcl is of course still limited. It doesn't come close to having call/cc or tail recursion. It doesn't have exact/inexact numbers, or most of the numerical tower. There is no memory management. Error reporting is spotty, and there is no error recovery.

About the book

I like writing documentation, and occasionally I'm good at it. I like to annotate the source code with bits of documentation, which I then extract and put together using tools like `awk`. It looks like this:



In the middle are a bunch of `.ans` files (ANnotated Source). These are written with Vim. From the TT tags of those, I extract a `.test` file. From the CB tags I extract `.tcl` source (or Scheme or C source, as the case might be). From all the tags except TT I extract formatted documentation in Markdown and LaTeX format. All these extractions are automated using `make`. Figures are made with Inkscape. I create a PDF document from the LaTeX source using TeXworks. On finishing up ConsTcl, it struck me that the documentation for this piece of software was fit for a book.

ConsTcl is at least 97% my own work, but I have ported one part (see page 116) from "Scheme 9 from Empty Space"[7] by Nils M Holm, a Lisp interpreter written in C and commented in a similar manner to this book.

[7] See `https://t3x.org/s9book/index.html`

One source which I spent a lot more time with was R5RS[8], *the* authoritative manual on Scheme. It says nothing about the implementation of the interpreter, but a lot on what features the language has and how they are supposed to work.

Another source of guidance and inspiration was "Lisp in Small Pieces" by Christian Queinnec. Yet another source was "An Introduction to Scheme and its Implementation"[9].

About the program listings

I have tried to write clear, readable code, but the page format forces me to shorten lines. I have used two-space indents instead of four-space, a smaller font, and broken off long lines with a \ at the end of the first line (a so-called 'tucked-in tail'). Neither of these measures improve readability, but the alternative is overwriting the margins. Not all broken lines have the \: some are broken inside a `{...}` block, and some right after a `[`.

About me

I'm a 60 year old former system manager who has been active in programming since 1979. Currently, and for some time, my language of choice is the rather marginal Tcl[10] (it's not even in the 100 most used languages). Tcl suits me, and there are

[8] Revised[5] Report on the Algorithmic Language Scheme, Scheme's standardization document

[9] See `https://docs.scheme.org/schintro/`

[10] See `https://www.tcl-lang.org/`

things that one can do in Tcl that one can't easily do in other languages. Lisp is a runner-up in my affections, a language that fascinates me but doesn't fit my brain very well (though I have written one large piece of software in AutoLisp, a CAD subsystem for designing drilling boxes).

In addition to my terms as programmer and system manager, I have worked as a teacher (teaching C/C++ in upper secondary school) and for a short while I wrote manuals for the department for information technology at the University of Skövde. I've also been active writing answers at question-and-answer sites on the web, mainly Stack Overflow.

About time

I'd like to thank

- my children and their lifemates for being awesome.
- my mother and my ex-wife for supporting the printing of the book financially.
- my sister and brother-in-law for being supportive.

And now let's journey into the interpreter.

Part I

The interpreter

1. Initial declarations

In this chapter there is mostly things I need to start working on the interpreter. Feel free to skim it, maybe coming back later to check up on things here.

First, I need to create the namespace that I will use for most identifiers:

```
1 namespace eval ::constcl {}
```

1.1 Utility commands

Next, some procedures that make my life as developer somewhat easier.

reg procedure

`reg` registers built-in procedures, special forms, and macros in the definitions register. That way I don't need to manually

keep track of and list procedures. The definitions register's contents will eventually get tranferred into the standard library (see page 304).

You can call `reg` with one parameter: *name*. *name* is a string that will eventually become the lookup symbol in the standard library. If you give two parameters, the first one is the *binding type*, either `special` or `macro`. The former registers special forms like `if` and `define`, and the latter registers macros like `and` or `when`. The second one is still the *name*.

There is also `regvar`, which registers variables. You pass *name* and *value* to it. There are only a couple of variables registered this way.

`reg` and `regvar` start out by checking if the definitions register (`defreg`) exists, and if not, they create it. Then they construct a *val*(ue) by concatenating a keyword (`VARIABLE`, `SPECIAL`, or `SYNTAX`) with a variation on *name* (or, in `regvar`'s case, *value*). Then they set an *index number* based on the current size of the `defreg`. Finally they insert the Tcl list of *name* and *val* under *index*.

reg (internal)	
?btype?	either 'special' or 'macro'
name	a Tcl string
Returns:	nothing

regvar (internal)	
name	a Tcl string
value	a value
Returns:	nothing

(This kind of box explains a few things about a procedure.

The last line shows what kind of value the procedure returns, if any. Above that are a number of lines that describe the parameters of the procedure, in order, by name and expected value type. If a parameter name is enclosed in ?...?, it means that the parameter is optional and can be left out.)

```
unset -nocomplain ::constcl::defreg

proc reg {args} {
  if {[llength $args] == 2} {
    lassign $args btype name
  } elseif {[llength $args] == 1} {
    lassign $args name
    set btype {}
  } else {
    error "wrong number of parameters\n([pn])"
  }
  if {![info exists ::constcl::defreg]} {
    set ::constcl::defreg [dict create]
  }
  switch $btype {
    special {
      set val [::list SPECIAL ::constcl::special-$name]
    }
    macro {
      set val [::list SYNTAX ::constcl::expand-$name]
    }
    default {
      set val [::list VARIABLE ::constcl::$name]
    }
  }
  set idx [dict size $::constcl::defreg]
  dict set ::constcl::defreg $idx [::list $name $val]
  return
}
```

Procedures, functions, and commands

I use all of these terms for the subroutines in ConsTcl. I try to stick with *procedure*, because that's the standard term in R5RS. Still, they usually pass useful values back to the caller, so technically they're *functions*. Lastly, I'm programming in Tcl here, and the usual term for these things is *commands* in Tcl.
And the *internal*/*public* distinction is possibly a misnomer. What it means is that *public* procedures can be called from Lisp code being interpreted, and the others cannot. They are for use in the infrastructure around the interpreter, including in implementing the *public* procedures. Another way to put it is that procedures registered by `reg` are *public* and those who aren't are *internal*.

```
31
32 proc regvar {name value} {
33   if {![info exists ::constcl::defreg]} {
34     set ::constcl::defreg [dict create]
35   }
36   set val [::list VARIABLE $value]
37   set idx [dict size $::constcl::defreg]
38   dict set ::constcl::defreg $idx [::list $name $val]
39   return
40 }
```

Predicates

By Scheme convention, predicates (procedures that return either `#t` or `#f`) have '?' at the end of their name. Some care is necessary when calling Scheme predicates from Tcl code (the Tcl `if` command expects 1 or 0 as truth values). Example:
`if {[atom? $x]} ...`
will not do, but
`if {[atom? $x] ne ${::#f}} ...`
("[atom? $x] not equal to false") works. Or see the `T` procedure.

atom? procedure

This one isn't just for my convenience: it's a standard procedure in Scheme. There are two kinds of data in Lisp: lists and atoms. Lists are collections of lists and atoms. Atoms are instances of types such as booleans, characters, numbers, ports, strings, symbols, and vectors. `Atom?` recognizes an atom by checking for membership in any one of the atomic types. It returns `#t` (true) if it is an atom, and `#f` (false) if not.

atom? (public)	
val	a value
Returns:	a boolean

```
41 reg atom?
42
43 proc ::constcl::atom? {val} {
44   foreach type {symbol number string
45       char boolean vector port eof} {
46     if {[$type? $val] eq ${::#t}} {
```

```
47        return ${::#t}
48      }
49    }
50    return ${::#f}
51  }
```

T procedure

The `T` procedure is intended to reduce the hassle of trying to make Lisp booleans work with Tcl conditions. The idea is to line the Tcl condition with `[T ...]` and have the Lisp expression inside `T`. `T` returns 0 if and only if the value passed to it is `#f`, and 1 otherwise. The procedure's name stands for 'truth of'.

Example:

```
if {[T [atom? $x]]} ...
```

T (internal)	
val	a value
Returns:	a Tcl truth value (1 or 0)

```
52  proc ::T {val} {
53    if {$val eq ${::#f}} {
54      return 0
55    } else {
56      return 1
57    }
58  }
```

assert procedure

`assert` signals an error if an assertion fails.

assert (internal)	
expr	a Tcl expression
Returns:	nothing

```
59 proc assert {expr} {
60   if {![uplevel [list expr $expr]]} {
61     error "Failed assertion [
62       uplevel [list subst $expr]]"
63   }
64 }
```

pairlis-tcl procedure

A Tcl version of the procedure in the Scheme base (see page 311).

pairlis-tcl (internal)	
lvals	a Lisp list of values
Returns:	a Lisp list of association pairs

```
65 proc ::constcl::pairlis-tcl {a b} {
66   if {[T [null? $a]]} {
67     parse {'()}
68   } else {
69     cons \
70       [cons [car $a] [car $b]] \
71       [pairlis-tcl [cdr $a] [cdr $b]]
72   }
73 }
```

pn procedure

`pn` stands for 'procedure name'. When called, tells the caller the name of its command. I use it for error messages so the error message can automagically tell the user which command failed.

pn (internal)	
Returns:	a Tcl string

```
74 proc ::pn {} {
75   namespace tail [lindex [info level -1] 0]
76 }
```

unbind procedure

`unbind` removes bindings from the environment they are bound in.

unbind (internal)	
syms	some symbols
Returns:	nothing

```
77 proc ::unbind {args} {
```

Try reading the value of `env` in the caller's context. If it succeeds, use that environment value; if it fails, use the global environment.

```
78   try {
79     uplevel [list subst \$env]
80   } on ok env {
```

```
81    } on error {} {
82      set env ::constcl::global_env
83    }
```

For each symbol given, check if it is bound in `env` or any of its linked environments except the null environment. If it is, unbind it there.

```
84    set syms $args
85    foreach sym $syms {
86      set e [$env find $sym]
87      if {$e ne "::constcl::null_env"} {
88        $e unbind $sym
89      }
90    }
91  }
```

typeof? procedure

`typeof?` looks at a value's type and reports if it is the same as the given type.

typeof? (internal)	
val	a value
type	a Tcl string
Returns:	a boolean

```
92  proc ::constcl::typeof? {val type} {
93    if {[info object isa typeof $val $type]} {
94      return ${::#t}
95    } else {
```

```
96      return ${::#f}
97    }
98  }
```

splitlist procedure

`splitlist` converts a Lisp list to a Tcl list with Lisp objects.

splitlist (internal)	
vals	a Lisp list of values
Returns:	a Tcl list of values

```
99  proc ::constcl::splitlist {vals} {
100   set result {}
101   while {[T [pair? $vals]]} {
102     lappend result [car $vals]
103     set vals [cdr $vals]
104   }
105   return $result
106 }
```

in-range procedure

This one is a little bit of both, a utility function that is also among the builtins in the library (it's not standard, though). It started out as a one-liner by Donal K Fellows, but has grown a bit since then to suit my needs.

The plan is to arrange a sequence of numbers, given one, two or three ConsTcl Number objects. If one is passed to the procedure, it is used as the end of the sequence: the sequence

will end just before it. If two numbers are passed, the first one becomes the start of the sequence: the first number in it. The second number will become the end of the sequence. If three numbers are passed, they become start, end, and step, i.e. how much is added to the current number to find next number in the sequence.

in-range (public)	
x	a number
?e?	a number
?t?	a number
Returns:	a Lisp list of numbers

```
reg in-range

proc ::constcl::in-range {x args} {
  set start 0
  set step 1
  switch [llength $args] {
    0 {
      set e $x
      set end [$e numval]
    }
    1 {
      set s $x
      lassign $args e
      set start [$s numval]
      set end [$e numval]
    }
    2 {
      set s $x
      lassign $args e t
      set start [$s numval]
      set end [$e numval]
```

```
128         set step [$t numval]
129       }
130     }
131     set res $start
132     while {$step > 0 && $end > [incr start $step] ||
133         $step < 0 && $end < [incr start $step]} {
134       lappend res $start
135     }
136     return [list {*}[lmap r $res {MkNumber $r}]]
137   }
```

error procedure

`error` is used to signal an error, with *msg* being a message string and the optional arguments being values to show after the message.

error (public)	
msg	a message string
?exprs?	some expressions
Returns:	-don't care-

```
138   reg error
139
140   proc ::constcl::error {msg args} {
141     set exprs $args
142     if {[llength $exprs]} {
143       set res [lmap expr $exprs {
144         $expr tstr
145       }]
146       ::append msg " (" [join $res] ")"
147     }
148     ::error $msg
```

```
149 }
```

check procedure

`check` does a check (typically a type check) on something and throws an error if it fails.

check (internal)	
cond	an expression
msg	a Tcl string
Returns:	nothing

```
150 proc ::constcl::check {cond msg} {
151   if {[uplevel $cond] eq ${::#f}} {
152     ::error [
153       uplevel [
154         ::list subst [
155           ::string trim $msg]]]
156   }
157 }
```

1.2 Testing commands

Testing gets easier if you have the software tools to manipulate and pick apart the testing data and actions. Short names reduce clutter in the test cases, at the cost of some readability.

pew procedure

`pew` was originally named `pep` after the sequence parse-eval-print. Now it's named for parse-eval-write. It reads an expression from a string, evals it, and writes the resulting value. It's the most common command in the test cases, since it allows me to write code directly in Scheme, get it evaled, and get to see proper Lisp output from it.

pew (internal)	
str	a Tcl string
?env?	an environment
Returns:	nothing

```
158 proc ::pew {str {env ::constcl::global_env}} {
159   ::constcl::write [
160     ::constcl::eval [parse $str] $env]
161 }
```

rew procedure

`rew` is the reading variant of `pew`. Instead of taking string input it takes a regular input port. It mattered more while the input library was being written.

rew (internal)	
port	an input port
?env?	an environment
Returns:	nothing

```
162 proc ::rew {port {env ::constcl::global_env}} {
```

```
163   ::constcl::write [
164     ::constcl::eval [
165       ::constcl::read $port] $env]
166 }
```

pw procedure

`pw` is a similar command, except it doesn't eval the expression. It just writes what is parsed. It is useful for tests when the evaluator can't (yet) evaluate the form, but I can still check if it gets read and written correctly.

pw (internal)	
str	a Tcl string
Returns:	nothing

```
167 proc ::pw {str} {
168   ::constcl::write [parse $str]
169 }
```

rw procedure

`rw` is the reading variant of `pw`. Instead of taking string input it takes a regular input port. The procedure just writes what is read.

rw (internal)	
?port?	an input port
Returns:	nothing

```
170 proc ::rw {args} {
171   ::constcl::write [::constcl::read {*}$args]
172 }
```

pe procedure

pe is also similar, but it doesn't write the expression. It just evaluates what is read. That way I get a value object which I can pass to another command, or pick apart in different ways.

pe (internal)	
str	a Tcl string
?env?	an environment
Returns:	a value

```
173 proc ::pe {str {env ::constcl::global_env}} {
174   ::constcl::eval [parse $str] $env
175 }
```

re procedure

re is like pe, but it reads from a regular port instead of from a string. It evaluates what is read.

re (internal)	
port	an input port
?env?	an environment
Returns:	a value

```
176 proc ::re {port {env ::constcl::global_env}} {
177   ::constcl::eval [::constcl::read $port] $env
178 }
```

parse procedure

parse only parses the input, returning an expression object.

parse (internal)	
str	a Tcl string
Returns:	an expression

```
179 proc ::parse {str} {
180   ::constcl::read [
181     ::constcl::MkStringInputPort $str]
182 }
```

e procedure

e is another single-action procedure, evaluating an expression and returning a value.

e (internal)	
expr	an expression
?env?	an environment
Returns:	a value

```
183 proc ::e {expr {env ::constcl::global_env}} {
184   ::constcl::eval $expr $env
185 }
```

w procedure

`w` is the third single-action procedure, printing a value and that's all.

w (internal)	
val	a value
Returns:	nothing

```
186 proc ::w {val} {
187   ::constcl::write $val
188 }
```

r procedure

`r` is an extra single-action procedure, reading from default input or from a port and returning an expression object.

r (internal)	
?port?	an input port
Returns:	an expression

```
189 proc ::r {args} {
190   ::constcl::read {*}$args
191 }
```

prw procedure

`prw` reads an expression, resolves defines, and writes the result. It was handy during the time I was porting the 'resolve local defines' section.

prw (internal)	
str	a Tcl string
Returns:	nothing

```
192 proc ::prw {str} {
193   set expr [parse $str]
194   set expr [::constcl::resolve-local-defines \
195     [::constcl::cdr $expr]]
196   ::constcl::write $expr
197 }
```

pxw procedure

pxw attempts to macro-expand whatever it reads, and writes the result.[11] Again, this command's heyday was when I was developing the macro facility.

pxw (internal)	
str	a Tcl string
?env?	an environment
Returns:	nothing

```
198 proc ::pxw {str {env ::constcl::global_env}} {
199   set expr [parse $str]
200   set op [::constcl::car $expr]
201   lassign [::constcl::binding-info $op $env] btype hinfo
202   if {$btype eq "SYNTAX"} {
203     set expr [$hinfo $expr $env]
204     ::constcl::write $expr
205   } else {
```

[11] I do know that 'expand' doesn't start with an 'x'.

```
206       ::error "not a macro"
207     }
208 }
```

1.3 Some small classes

Base class

The `Base` class is base class for most of the type classes.

```
209 catch { ::constcl::Base destroy }
210
211 oo::abstract create ::constcl::Base {
```

The `mkconstant` method is a dummy method that can be called when the instance is part of an immutable structure. Classes that change their state when this method is called will override it.

(concrete instance) mkconstant (internal)	
Returns:	nothing

```
212   method mkconstant {} {}
```

The `write` method is used by the `write` standard procedure to print the external representation of an instance.

(concrete instance) write (internal)	
port	an output port
Returns:	nothing

```
213   method write {port} {
214     $port put [my tstr]
215   }
```

The `display` method is used by the `display` standard procedure to print the external representation or a human-readable version of an instance. In the latter case the method will be overridden.

(concrete instance) display (internal)	
port	an output port
Returns:	nothing

```
216   method display {port} {
217     my write $port
218   }
```

The `show` method yields the external representation of the instance as a string.

(concrete instance) show (internal)	
Returns:	a string

```
219   method show {} {
220     ::constcl::MkString [my tstr]
221   }
```

The `tstr` method yields the external representation of the instance as a Tcl string. It is used by error messages and the `write` method. Should be overridden by a concrete class.

(concrete instance) tstr (internal)	
Returns:	a Tcl string

```
222   method tstr {} {
223     return "#<base>"
224   }
```

The `unknown` method responds to calls to undefined methods. It produces a suitable error message.

(concrete instance) unknown (internal)	
name	a Tcl string
args	some arguments
Returns:	nothing

```
225   method unknown {name args} {
226     switch $name {
227       car - cdr - set-car! -
228       set-cdr {
229         ::error "PAIR expected"
230       }
231       numval {
232         ::error "NUMBER expected"
233       }
234     }
235   }
236 }
```

Dot class

The `Dot` class is a helper class for the parser.

```
237 oo::class create ::constcl::Dot {
238   superclass ::constcl::Base
```

```
239   method tstr {} {
240     format "."
241   }
242 }
```

dot? procedure

`dot?` is a type predicate that checks for membership in the type `Dot`.

dot? (internal)	
val	a value
Returns:	a boolean

```
243 proc ::constcl::dot? {val} {
244   typeof? $val "Dot"
245 }
```

EndOfFile class

The `EndOfFile` class is for end-of-file conditions.

```
246 oo::singleton create ::constcl::EndOfFile {
247   superclass ::constcl::Base
248   method tstr {} {
249     format "#<end-of-file>"
250   }
251 }
```

eof? procedure

`eof?` is a type predicate that recognizes the end-of-file object.

eof? (internal)	
val	a value
Returns:	a boolean

```
252 proc eof? {val} {
253   if {$val eq ${::#EOF}} {
254     return ${::#t}
255   } else {
256     return ${::#f}
257   }
258 }
```

NIL class

The `NIL` class has one instance: the empty list called `#NIL`.

```
259 oo::singleton create ::constcl::NIL {
260   superclass ::constcl::Base
261   method tstr {} {
262     return "()"
263   }
264 }
```

null? procedure

The `null?` standard predicate recognizes the empty list.

null? (public)	
val	a value
Returns:	a boolean

```
reg null?

proc ::constcl::null? {val} {
  if {$val eq ${::#NIL}} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

Undefined class

The `Undefined` class is for undefined things. It was created to facilitate porting of code from 'Scheme 9 from Empty Space'.

```
oo::singleton create ::constcl::Undefined {
  superclass ::constcl::Base
  method tstr {} {
    format "#<undefined>"
  }
}
```

Unspecified class

The `Unspecified` class is for unspecified things. Also a S9fES support class.

```
280 oo::singleton create ::constcl::Unspecified {
281   superclass ::constcl::Base
282   method tstr {} {
283     format "#<unspecified>"
284   }
285 }
```

2. Input

The first thing an interpreter must be able to do is to take in the user's code and data (*input*) and make it fit to be evaluated. This is handled by the `read` procedure.

2.1 Input and ports

The main challenge in taking in code and data is to determine from which device the input will come. Possible alternatives are the keyboard, an input file, a string buffer, etc. To make input streamlined, the various kinds of devices are abstracted into *ports*.

The procedure `read` represents the interpreter's main input facility.

read procedure

One can pass a port to `read` in which case `read` sets the current input port temporarily to the provided port. If no port is

passed, `read` uses the default standard input port (usually the keyboard[12]).

read (public)	
?port?	an input port
Returns:	an expression

```
286 reg read
287
288 proc ::constcl::read {args} {
289   set c {}
290   set unget {}
291   set oldport $::constcl::Input_port
292   if {[llength $args]} {
293     lassign $args port
294     set ::constcl::Input_port $port
295   }
296   set expr [read-expr]
297   set ::constcl::Input_port $oldport
298   return $expr
299 }
```

2.2 Parsing

Making input fit to be evaluated is a more complex procedure, and most of the procedures in this section deal with this conversion, which is also known as *parsing*. To make parsing possible, the input must be encoded in a way that makes the type of input obvious and consistent.

[12] which doesn't work in a Windows windowing environment, e.g. wish or tkcon. repl does work in those, though. Input works in tclsh on Windows.

Ports

Ports are an abstraction of the input or output mechanism. An input port can be connected to standard input (the keyboard) or a file opened for input or a string input buffer where the complete available input is laid out before reading starts. Regardless of what kind of input port it is, one can read characters from it until it runs out and signals end-of-file. Likewise, an output port, regardless of whether it's the standard output–the screen–or a file opened for output, will receive characters sent to it.

Parsing[13], or syntactic analysis, is analyzing a sequence of letters, digits, and other characters, a piece of text conforming to the rules of *external representation*. The result of parsing is an *expression* in *internal representation*.

As a simple example of external representation, `99` denotes a number, while `"99"` denotes a string. The `read` procedure takes in input character by character, matching each character against a fitting external representation. When done, it creates a ConsTcl object, which is the internal representation of an expression. The object can then be passed to the evaluator.

`99` is parsed into a Number object, while `"99"` is parsed into a String object. `'(99 "99")` is parsed into a quoted Pair structure with two elements, a Number object and a String object.

Once input has been parsed into an expression, the expression can be evaluated using the `eval` procedure (see page 95)

[13]See `https://en.wikipedia.org/wiki/Parsing`

External representation

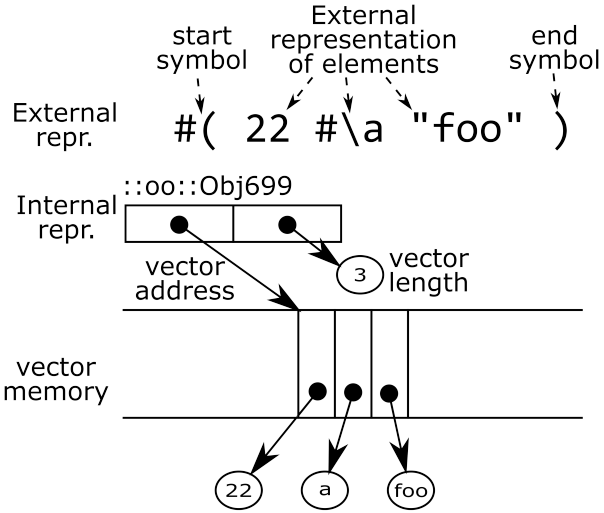
The external representation is a 'recipe' for an expression that expresses it in a unique and consistent way.
For example, the external representation for a vector is a pound sign (`#`), a left parenthesis (`(`), the external representation for some values, and a right parenthesis (`)`). When the reader/-parser is working its way through input, a `#(` symbol signals that a vector structure is being read. A number of subexpressions for the elements of the vector follow, and then a closing parenthesis `)` signals that the vector is done. The elements are saved in vector memory and the vector gets the address to the first element and the number of elements.
Some types of data and external representation:
String: `"abc"`
Character: `#\c`
Vector: `#(99 "abc")`
List (stored as a Pair structure): `(1 2)` or `[3 4]`
Number: `99`
Identifier: `abc`

or printed using the `write` procedure (see page 125).

A non-quoted list is parsed into a structure of Pair objects and evaluates to the result of the execution of the components of the list (the operator and the operands).

Example (running in `tclsh`):

```
% ::constcl::read
(+ 2 3)
```



Vector representation

```
::oo::Obj491
```

Here, `read` read and parsed the external representation of a list with three elements, +, 2, and 3. It produced the expression that has an internal representation labeled `::oo::Obj491` (the number has no significance other than to identifiy the object: it will be different each time the code is run).

Printing the object returns it to external representation:

```
% ::constcl::write ::oo::Obj491
(+ 2 3)
```

Evaluating the object creates a new expression (the result of applying the operator to the operands):

```
% ::constcl::eval ::oo::Obj491
::oo::Obj494
```

Printing it shows the result:

```
% ::constcl::write ::oo::Obj494
5
```

Fortunately, we don't *have* to work at such a low level. We can use the `repl` instead:

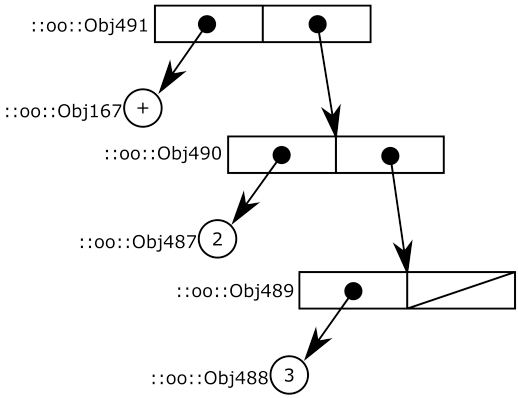
```
ConsTcl> (+ 2 3)
5
```

Then, parsing, evaluation, and writing goes on in the background and the internal representations of expressions and values are hidden.

Anyway, the figure shows what it really looks like. `::oo::Obj491` was just the head of the list.

(The parts of the list can be inspected using the `car` and `cdr` procedures (see page 242).)

For the rest of subsection, procedures that implement parsing.



The internal structure of the expression

The `read-` procedures parse their input and produce Cons-Tcl objects.

Reader procedures specialize in reading a certain kind of input, except for `read-expr` which reads them all (with a little help).

read-expr procedure

The `read-expr` procedure reads the first available character from the input port. Based on that character it delegates to one of the more detailed readers, producing an expression of the cor-

responding kind. A Tcl character value can be passed to it: that character will be used first before reading from the input. If end of file is encountered before an expression can be read in full, the procedure returns end of file (`#EOF`). Shares the variables `c` and `unget` with its caller.

read-expr (internal)	
?char?	a Tcl character
Returns:	an expression or end of file

```
300 proc ::constcl::read-expr {args} {
301   upvar c c unget unget
302   if {[llength $args]} {
303     lassign $args c
304   } else {
305     set c [readchar]
306   }
307   set unget {}
308   read-eof $c
309   if {[::string is space $c] || $c eq ";"} {
310     skip-ws
311     read-eof $c
312   }
313   switch -regexp $c {
314     {\"}          { read-string-expr }
315     {\#}          { read-pound }
316     {\'}          { read-quoted-expr }
317     {\(}          { read-pair-expr ")" }
318     {\+} - {\-}   { read-plus-minus $c }
319     {\,}          { read-unquoted-expr }
320     {\.} {
321         set x [Dot new]; set c [readchar]; set x
322     }
323     {\[}          { read-pair-expr "\]" }
324     {\`}          { read-quasiquoted-expr }
```

```
325       {\d}           { read-number-expr $c }
326       {^$}           { return #EOF }
327       {[[:graph:]]} { read-identifier-expr $c }
328       default {
329         read-eof $c
330         ::error "unexpected character ($c)"
331       }
332     }
333   }
```

read-character-expr procedure

`read-character-expr` is activated by `read-pound` when that procedure finds a backslash in the input stream (pound-backslash is the external representation prefix for characters). It reads one or more characters to produce a character expression and return a Char object (see page 184). Shares the variables `c` and `unget` with its caller.

read-character-expr (internal)	
Returns:	a character or end of file

```
334   proc ::constcl::read-character-expr {} {
335     upvar c c unget unget
336     set name "#\\"
```

A character name can be one or more characters long. Accept the first character if it is a graphic character.

```
337     set c [readchar]
338     read-eof $c
```

```
339   if {[::string is graph $c]} {
340     ::append name $c
```

Keep adding to the name as long as the input is an alphabetic character.

```
341     set c [readchar]
342     while {[::string is alpha $c]} {
343       ::append name $c
344       set c [readchar]
345     }
346   }
```

Check if we have a valid character name.

```
347   check {valid-char? $name} {
348       Invalid character constant $name
349   }
```

Make and return a character object.

```
350   set expr [MkChar $name]
351   read-eof $expr
352   return $expr
353 }
```

read-identifier-expr procedure

`read-identifier-expr` is activated for 'anything else', and takes in characters until it finds whitespace or a delimiter character. If it is passed one or more characters it will use them before

consuming any from input. It checks the input against the rules for identifiers, accepting or rejecting it with an error message. It returns a Symbol object (see page 282). Shares the variables `c` and `unget` with its caller.

read-identifier-expr (internal)	
?chars?	some Tcl characters
Returns:	a symbol

```
354 proc ::constcl::read-identifier-expr {args} {
355   upvar c c unget unget
356   set unget {}
```

If one or more characters have been passed to the procedure, join them together and store them in `c`. Otherwise, read a character from input.

```
357   if {[llength $args]} {
358     set c [join $args {}]
359   } else {
360     set c [readchar]
361   }
362   read-eof $c
363   set name {}
```

Add the contents of `c` to `name` as long as the character is graphic and not a delimiter or `#EOF`.

```
364   while {[::string is graph -strict $c]} {
365     if {$c eq "#EOF" || [T [delimiter? $c]]} {
366       break
367     }
```

```
368       ::append name $c
369       set c [readchar]
370       # do not check for EOF here
371     }
```

If the last character read is a delimiter, unget it.

```
372     if {[T [delimiter? $c]]} {
373       set unget $c
374     }
```

Check if the name is a valid identifier, and create and return a symbol object.

```
375     # idcheck throws error if invalid identifier
376     idcheck $name
377     return [S $name]
378   }
```

read-number-expr procedure

`read-number-expr` reads numerical input, both integers and floating point numbers. It is activated by `read-expr` or `read-plus-minus` if they encounter digits, and it actually takes in anything that at least starts out like a number. It stops at whitespace or a delimiter character, and then it accepts or rejects the input by comparing it to a Tcl double. It returns a Number object (see page 153). Shares the variables `c` and `unget` with its caller.

read-number-expr (internal)	
?char?	a Tcl character
Returns:	a number or end of file

```
379 proc ::constcl::read-number-expr {args} {
380   upvar c c unget unget
381   set unget {}
```

If a character has been passed to the procedure, store it in `c`. Otherwise, read a character from input.

```
382   if {[llength $args]} {
383     lassign $args c
384   } else {
385     set c [readchar]
386   }
387   read-eof $c
```

Add the contents of `c` to `num` as long as the character isn't space, `#EOF`, or a delimiter.

```
388   while {![T [interspace? $c]] && $c ne "#EOF" &&
389       ![T [delimiter? $c]]} {
390     ::append num $c
391     set c [readchar]
392   }
```

If the last character read is a delimiter, unget it.

```
393   if {[T [delimiter? $c]]} {
394     set unget $c
395   }
```

Check if the contents of num is a valid number, and create and return a number object.

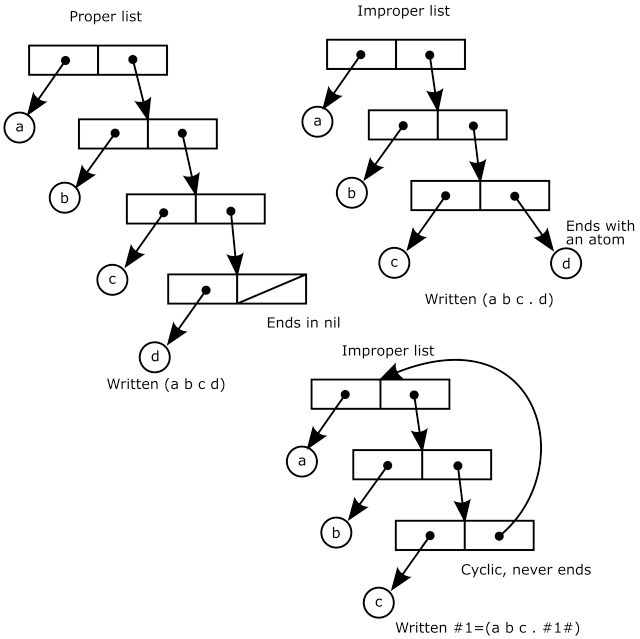
```
396   check {::string is double -strict $num} {
397       Invalid numeric constant $num
398   }
399   set expr [N $num]
400   return $expr
401 }
```

read-pair-expr procedure

The read-pair-expr procedure reads everything between two matching parentheses, or, as the case might be, brackets. It produces either an empty list, or a possibly recursive structure of Pair objects (see page 233), either a proper list (one that ends in NIL), or an improper one (one that has an atom as its last member). Note that read-pair-expr can't read a cyclic improper list. Shares the variables c and unget with its caller.

read-pair-expr (internal)	
char	the terminating paren or bracket
Returns:	a structure of pair expressions or end of file

```
402 proc ::constcl::read-pair-expr {char} {
403   upvar c c unget unget
404   set unget {}
405   set expr [read-pair $char]
406   read-eof $expr
407   if {$c ne $char} {
```



A proper list and two improper ones.

```
408       if {$char eq ")"} {
409         ::error \
410           "Missing right paren. ($c)."
411       } else {
412         ::error \
413           "Missing right bracket ($c)."
```

```
414       }
415     } else {
416       set unget {}
417       set c [readchar]
418     }
419     return $expr
420   }
```

read-pair procedure

`read-pair` is a helper procedure that does the heavy lifting in reading a pair structure. First it checks if the list is empty, returning `NIL` in that case. Otherwise it reads the first element in the list and then repeatedly the rest of them. If it reads a Dot object, the following element to be read is the tail end of an improper list. When `read-pair` has reached the ending parenthesis or bracket, it "conses up" the elements starting from the last, and returns the head of the list. Shares the variables `c` and `unget` with its caller.

read-pair (internal)	
char	the terminating paren or bracket
Returns:	a structure of pair expressions or end of file

```
421 proc ::constcl::read-pair {char} {
422   upvar c c unget unget
```

If the first non-space character is the ending parenthesis or bracket, return an empty list.

```
423   set c [readchar]
```

```
424   read-eof $c
425   if {[T [find-char? $char]]} {
426     return ${::#NIL}
427   }
```

Read an expression and put it in the result list. Tentatively set the end of the list to `#NIL`.

```
428   set a [read-expr $c]
429   set res $a
430   skip-ws
431   set prev ${::#NIL}
```

As long as the ending parenthesis or bracket isn't found, read an expression into `x`. If it is a dot, read another expression and set the end of the list to it. Otherwise, append the expression in `x` to the result list.

```
432   while {![T [find-char? $char]]} {
433     set x [read-expr $c]
434     skip-ws
435     read-eof $c
436     if {[T [dot? $x]]} {
437       set prev [read-expr $c]
438       skip-ws
439       read-eof $c
440     } else {
441       lappend res $x
442     }
443   }
```

Reverse the result list and construct pairs from each item and the current end of the list. Return the final end of the list.

```
444   foreach r [lreverse $res] {
445     set prev [cons $r $prev]
446   }
447   return $prev
448 }
```

read-plus-minus procedure

`read-plus-minus` is called when a plus or minus is found in the input stream. The plus or minus character is passed to it. If the next character is a digit, it delegates to the number reader. If it is a space character or a delimiter, it returns a + or − symbol. Otherwise, it delegates to the identifier reader. Shares the variables `c` and `unget` with its caller.

read-plus-minus (internal)	
char	a Tcl character
Returns:	either the symbols + or - or a number or end of file

```
449 proc ::constcl::read-plus-minus {char} {
450   upvar c c unget unget
451   set unget {}
452   set c [readchar]
453   read-eof $c
```

If the first character read is a digit, read a number. If the character passed to the procedure was a minus sign, make the number negative.

```
454   if {[::string is digit -strict $c]} {
455     set expr [read-number-expr $c]
456     read-eof $expr
457     if {$char eq "-"} {
458       set expr [- $expr]
459     }
460     return $expr
```

If the first character read is a space or delimiter character, return a + or − symbol, depending on the character passed to the procedure.

```
461   } elseif {[T [interspace? $c]] ||
462       [T [delimiter? $c]]} {
463     if {$char eq "+"} {
464       return [S "+"]
465     } else {
466       return [S "-"]
467     }
```

Otherwise, read an identifier.

```
468   } else {
469     set expr [read-identifier-expr $char $c]
470     read-eof $expr
471     return $expr
472   }
473 }
```

read-pound procedure

`read-pound` is activated by `read-expr` when it reads a pound sign (`#`). It in turn either delegates to the vector or character reader, or returns boolean literals. Shares the variables `c` and `unget` with its caller.

read-pound (internal)	
Returns:	a vector, boolean, or character value or end of file

```
474 proc ::constcl::read-pound {} {
475   upvar c c unget unget
476   set unget {}
477   set c [readchar]
478   read-eof $c
479   switch $c {
480     (     { set expr [read-vector-expr] }
481     t     { if {[T [read-end?]]} {set expr ${::#t}} }
482     f     { if {[T [read-end?]]} {set expr ${::#f}} }
483     "\\" { set expr [read-character-expr] }
484     default {
485       ::error "Illegal #-literal: #$c"
486     }
487   }
488   return $expr
489 }
```

read-quasiquoted-expr procedure

`read-quasiquoted-expr` is activated when there is a backquote (`‘`) in the input stream. It reads an entire expression and returns

it wrapped in `quasiquote`. Shares the variables `c` and `unget` with its caller.

read-quasiquoted-expr (internal)	
Returns:	an expr. wr. in the quasiquote symbol or end of file

```
490 proc ::constcl::read-quasiquoted-expr {} {
491   upvar c c unget unget
492   set unget {}
493   set expr [read-expr]
494   read-eof $expr
495   make-constant $expr
496   return [list [S quasiquote] $expr]
497 }
```

read-quoted-expr procedure

`read-quoted-expr` is activated by `read-expr` when reading a single quote (’). It then reads an entire expression beyond that, returning it wrapped in a list with `quote`. The quoted expression is made constant. Shares the variables `c` and `unget` with its caller.

read-quoted-expr (internal)	
Returns:	an expression wrapped in the quote symbol or end of file

```
498 proc ::constcl::read-quoted-expr {} {
499   upvar c c unget unget
500   set unget {}
501   set expr [read-expr]
502   read-eof $expr
```

```
503   make-constant $expr
504   return [list [S quote] $expr]
505 }
```

read-string-expr procedure

`read-string-expr` is activated by `read-expr` when it reads a double quote. It collects characters until it reaches another (unescaped) double quote. To have double quotes in the string, escape them with backslash (which also means that backslashes have to be escaped with backslash). A backslash+n pair of characters denotes a newline (this is an extension). It then returns a string expression–an immutable String object (see page 257). Shares the variables `c` and `unget` with its caller.

read-string-expr (internal)	
Returns:	a string

```
506 proc ::constcl::read-string-expr {} {
507   upvar c c unget unget
508   set str {}
509   set c [readchar]
510   read-eof $c
```

As long as the input isn't a double quote or end-of-file, add it to `str`.

```
511   while {$c ne "\"" && $c ne "#EOF"} {
```

If the input is a backslash, add it to `str` and read another character. In this way escaped double quotes are bypassed.

```
512     if {$c eq "\\"} {
513       ::append str $c
514       set c [readchar]
515       read-eof $c
516     }
517     ::append str $c
518     set c [readchar]
519   }
```

If the last read character is end-of-file, the ending double quote was missing.

```
520   if {$c eq "#EOF"} {
521     error "bad string (no ending double quote)"
522   }
523   set c [readchar]
```

Create and return an immutable string object.

```
524   set expr [MkString $str]
525   make-constant $expr
526   return $expr
527 }
```

read-unquoted-expr procedure

When a comma is found in the input stream, `read-unquoted-expr` is activated. If it reads an at-sign (`@`) it selects the symbol `unquote-splicing`, otherwise it selects the symbol `unquote`. Then it reads an entire expression and returns it wrapped in the

selected symbol. Both of these expressions are only supposed to occur inside a quasiquoted expression. Shares the variables `c` and `unget` with its caller.

read-unquoted-expr (internal)	
Returns:	an expr. wr. in the unquote/-splicing symbol or end of file

```
528 proc ::constcl::read-unquoted-expr {} {
529   upvar c c unget unget
530   set unget {}
531   set c [readchar]
532   read-eof $c
533   if {$c eq "@"} {
534     set symbol "unquote-splicing"
535     set expr [read-expr]
536   } else {
537     set symbol "unquote"
538     set expr [read-expr $c]
539   }
540   read-eof $expr
541   return [list [S $symbol] $expr]
542 }
```

read-vector-expr procedure

`read-vector-expr` is activated by `read-pound`. It reads a number of expressions until it finds an ending parenthesis. It produces a vector expression and returns a Vector object (see page 288). Shares the variables `c` and `unget` with its caller.

read-vector-expr (internal)	
Returns:	a vector or end of file

```
proc ::constcl::read-vector-expr {} {
  upvar c c unget unget
  set res {}
  set last {}
  set c [readchar]
  while {$c ne "#EOF" && $c ne ")"} {
```

Read an expression, put it in an element constructed as a pair with the expression and `#NIL`, and affix the element to the result list.

```
    set e [read-expr $c]
    read-eof $e
    set elem [cons $e ${::#NIL}]
    if {$res eq {}} {
      set res $elem
    } else {
      set-cdr! $last $elem
    }
    set last $elem
    skip-ws
    read-eof $c
  }
```

Report missing ending parenthesis.

```
  if {$c ne ")"} {
    ::error "Missing right paren. ($c)."
  }
  set unget {}
  set c [readchar]
```

Create and return an immutable vector object.

```
566   set expr [MkVector $res]
567   $expr mkconstant
568   return $expr
569 }
```

2.3 Input helper procedures

In this subsection, some procedures which are used by the reading/parsing procedures.

make-constant procedure

The `make-constant` helper procedure is called to set expressions to constants when read as a literal.

make-constant (internal)	
val	a value
Returns:	nothing

```
570 proc ::constcl::make-constant {val} {
571   if {[T [pair? $val]]} {
572     $val mkconstant
573     make-constant [car $val]
574     make-constant [cdr $val]
575   } elseif {[T [null? $val]]} {
576     return
577   } else {
578     $val mkconstant
579     return
580   }
581 }
```

interspace? procedure

The `interspace?` helper procedure recognizes whitespace between value representations.

interspace? (internal)	
c	a Tcl character
Returns:	a boolean

```
proc ::constcl::interspace? {c} {
  if {[::string is space $c]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

delimiter? procedure

The `delimiter?` helper procedure recognizes delimiter characters between value representations.

delimiter? (internal)	
c	a Tcl character
Returns:	a boolean

```
proc ::constcl::delimiter? {c} {
  if {$c in {( ) ; \" ’ ‘ | [ ] \{ \}}} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

valid-char? procedure

The `valid-char?` helper procedure compares a potential character constant to the valid kinds.

valid-char? (internal)	
name	a Tcl string
Returns:	a boolean

```
596 proc ::constcl::valid-char? {name} {
597   if {[regexp {(?i)^#\\([[:graph:]]|space|newline)$} \
598       $name]} {
599     return ${::#t}
600   } else {
601     return ${::#f}
602   }
603 }
```

readchar procedure

`readchar` reads one character from the `unget` store if it isn't empty or else from the input port. If the input is at end-of-file, an `#EOF` object is returned. Shares the variable `unget` with its caller.

readchar (internal)	
Returns:	a Tcl character or end of file

```
604 proc ::constcl::readchar {} {
605   upvar unget unget
606   if {$unget ne {}} {
607     set c $unget
608     set unget {}
```

```
609   } else {
610     set c [$::constcl::Input_port get]
611     if {[$::constcl::Input_port eof]} {
612       return #EOF
613     }
614   }
615   return $c
616 }
```

find-char? procedure

`find-char?` reads ahead through whitespace to find a given character. It returns `#t` if it has found the character, and `#f` if it has stopped at some other character. Sets `unget` to the character it stopped at. Returns end of file if eof is encountered. Shares the variables `c` and `unget` with its caller.

find-char? (internal)	
char	a Tcl character
Returns:	a boolean or end of file

```
617 proc ::constcl::find-char? {char} {
618   upvar c c unget unget
619   # start with stored c
620   while {[::string is space -strict $c]} {
621     # this order seems strange but works
622     set c [readchar]
623     read-eof $c
624     set unget $c
625   }
626   expr {($c eq $char) ? ${::#t} : ${::#f}}
627 }
```

read-end? procedure

`read-end?` reads one character and returns `#t` if it is an interspace character or a delimiter character, or `#EOF` if at end of file. Otherwise it returns `#f`. It ungets the character before returning, unless the character was interspace or end-of-file. Shares the variables `c` and `unget` with its caller.

read-end? (internal)	
Returns:	a boolean or end of file

```
628 proc ::constcl::read-end? {} {
629   upvar c c unget unget
630   set c [readchar]
631   if {[T [interspace? $c]]} {
632     return ${::#t}
633   } elseif {[T [delimiter? $c]]} {
634     set unget $c
635     return ${::#t}
636   } elseif {$c eq "#EOF"} {
637     return #EOF
638   } else {
639     set unget $c
640     return ${::#f}
641   }
642 }
```

skip-ws procedure

`skip-ws` skips whitespace and comments (the ; to end of line kind). It leaves the first character not to be skipped in `c` and also ungets it. Shares the variables `c` and `unget` with its caller.

skip-ws (internal)	
Returns:	nothing

```
643 proc ::constcl::skip-ws {} {
644   upvar c c unget unget
645   while true {
646     switch -regexp $c {
647       {[[:space:]]} {
648         set c [readchar]
649       }
650       {;} {
651         while {$c ne "\n" && $c ne "#EOF"}  {
652           set c [readchar]
653         }
654       }
655       default {
656         set unget $c
657         return
658       }
659     }
660   }
661 }
```

read-eof procedure

`read-eof` checks a number of presumed characters for possible end-of-file objects. If it finds one, it returns *from its caller* with the EOF value.

read-eof (internal)	
chars	some characters
Returns:	nothing

```
662 proc ::constcl::read-eof {args} {
663   set chars $args
664   foreach char $chars {
665     if {$char eq "#EOF"} {
666       return -level 1 -code return #EOF
667     }
668   }
669 }
```

3. Evaluation

The second thing an interpreter must be able to do is to *evaluate* expressions, that is reduce them to *normal form*. As an example, 2 + 6 and 8 are two expressions that have the same value, but the latter is in normal form (can't be reduced further) and the former is not.

To be able to evaluate every kind of expression, a structured approach is needed. Lisp has nine syntactic forms, each with its own syntax, and each with its own process of evaluation.

variable reference Syntax: a symbol. Process: variable lookup (see page 62).

constant literal Syntax: a string, character, boolean, or number. Process: take the value (see page 64).

quotation Syntax: `(quote datum)`. Process: take the datum (see page 65).

conditional Syntax: `if`, `case`, or `cond` expression. Process: depends on which syntax (see page 66).

sequence Syntax: `(begin expression ...)`. Process: evaluate all expressions, take value of the last (see page 74).

definition Syntax: `(define var val)`. Process: bind a variable to a location, store the value there (see page 76).

assignment Syntax: `(set! var val)`. Process: take a bound variable, store the value to its location (see page 79).

procedure definition Syntax: `(lambda formals body)`. Process: take formals and body and apply lambda to get a procedure value (see page 80).

procedure call Syntax: `(operator operand ...)`. Process: invoke operator on operands (see page 83).

The evaluator recognizes the syntax of the expression and chooses the appropriate process to evaluate it. How this happens for the nine syntactic forms will be described in the following sections.

A word about *environments*: an environment is where evaluating code keeps track of things. This is why most of the procedures in this chapter get a reference to an environment when they are called. More about environments very soon (see page 93).

3.1 Variable reference

Example: $\mathtt{r} \Rightarrow 10$ *(a symbol* `r` *is evaluated to what it's bound to)*

A variable is about a symbol, a location in the environment, and a value. The symbol is *bound* to the location, and the value is stored there. When an expression consists of the symbol, the evaluator does *lookup* and finds the value.

This is handled by the helper procedure `lookup`. It (or rather, the helper function `binding-info`, which it calls) searches the environment chain (see page 93) for the symbol, and returns the value stored in the location it is bound to. It is an error to do lookup on an unbound symbol, or a symbol that is bound for some other purpose, such as being a keyword or a macro.

Syntax: *symbol*

lookup procedure

lookup (internal)	
sym	a symbol
env	an environment
Returns:	a value

```
670 proc ::constcl::lookup {sym env} {
671   lassign [binding-info $sym $env] type value
672   if {$type eq "VARIABLE"} {
673     return $value
674   } else {
675     error "not a variable name" $sym
676   }
677 }
```

3.2 Constant literal

Example: `99` ⇒ 99 (a number evaluates to itself)

Not just numbers but booleans, characters, and strings evaluate to themselves, to their innate value. Because of this, they are called self-evaluating or autoquoting types (see next section).

Syntax: *number* | *string* | *character* | *boolean*

self-evaluating? procedure

Only numeric, string, character, and boolean constants evaluate to themselves. This procedure returns `#t` if the given value is a self-evaluating value, and `#f` otherwise.

self-evaluating? (internal)	
val	a value
Returns:	a boolean

```
678 proc ::constcl::self-evaluating? {val} {
679   if {[T [number? $val]] ||
680     [T [string? $val]] ||
681     [T [char? $val]] ||
682     [T [boolean? $val]]} {
683     return ${::#t}
684   } else {
685     return ${::#f}
686   }
687 }
```

3.3 Quotation

Example: `(quote r)` ⇒ `r` *(quotation makes the symbol evaluate to itself, like a constant)*

According to the rules of variable reference, a symbol evaluates to its stored value. Sometimes one wishes to use the symbol itself as a value. That is partly what quotation is for. `(quote x)` evaluates to the symbol `x` itself and not to any value that might be stored under it. This is so common that there is a shorthand notation for it: `'x` is interpreted as `(quote x)` by the Lisp reader (see page 29). The argument of `quote` may be any external representation (see page 32) of a Lisp object. In this way, for instance a vector or list constant can be introduced in the program text.

quote special form

Syntax: (**quote** *datum*)

The `quote` special form is expanded by `special-quote`.

special-quote (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
688 reg special quote
689
690 proc ::constcl::special-quote {expr env} {
691   cadr $expr
692 }
```

3.4 Conditional

Example: `(if (> 99 100) (* 2 2) (+ 2 4))` $\Rightarrow$ *6*

The conditional form `if` takes three expressions. The first, the *condition*, is evaluated first. If it evaluates to true, i.e. anything other than the value `#f` (false), the second expression (the *consequent*) is evaluated and the value returned. Otherwise, the third expression (the *alternate*) is evaluated and the value returned. One of the two latter expressions will be evaluated, and the other will remain unevaluated. The *alternate* can be omitted.

if special form

Syntax: (**if** *condition consequent* ?*alternate*?)

The `if` special form is expanded by `special-if`.

special-if (internal)	
expr	an expression
env	an environment
Returns:	a value

```
693 reg special if
694
695 proc ::constcl::special-if {expr env} {
696   set args [cdr $expr]
697   if {[T [null? [cddr $args]]]} {
698     if {[T [eval [car $args] $env]]} \
699       {eval [cadr $args] $env}
700   } else {
701     if {[T [eval [car $args] $env]]} \
702       {eval [cadr $args] $env} \
```

```
703         {eval [caddr $args] $env}
704     }
705 }
```

case special form

`case` is another conditional form. It implements a multi-choice where a single expression selects between alternatives. The body of the `case` form consists of a key-expression and a number of clauses. Each clause has a list of values and a body. If the key-expression evaluates to a value that occurs in one of the value-lists (considered in order), that clause's body is evaluated and all other clauses are ignored.

Syntax: (**case** *key clause* ...)
where each *clause* has the form
((*datum* ...) *expression* ...)
The last clause may have the form
(**else** *expression* ...)
Example:

```
(case 'c
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant))     ==> consonant
```

The `case` special form is expanded by `special-case`. It expands to `'()` if there are no clauses (left), and to nested `if` constructs if there are some.

caar, cadr, cdar, and the rest

The `do-case` procedure uses extensions of the `car/cdr` operators like `caar` and `cdar`. `car/cdr` notation gets really powerful when combined to form operators from `caar` to `cddddr`. One can read `caar L` as 'the first element of the first element of L, implying that the first element of `L` is a list. `cdar L` is 'the rest of the elements of the first element of L, and `cadr L` is 'the first element of the rest of the elements of `L` or in layman's terms, the second element of `L`.

special-case procedure

special-case (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
706 reg special case
707
708 proc ::constcl::special-case {expr env} {
709   set tail [cdr $expr]
710   set expr [do-case [car $tail] [cdr $tail] $env]
711   eval $expr $env
712 }
```

do-case procedure

Quasiquote

In this and many other special form and macro expanders I use a quasiquote construct to lay out how the form is to be expanded. A quasiquote starts with a backquote (`) instead of the single quote that precedes regular quoted material. A quasiquote allows for 'unquoting' of selected parts: this is notated with a comma (,). `` `(foo ,bar baz) `` is very nearly the same as `('foo bar 'baz)`. In both cases `foo` and `baz` are constants while `bar` is a variable which will be evaluated. Like in `do-case` here, a quasiquote serves well as a templating mechanism. The variables in the quasiquote need to be a part of the environment in which the quasiquote is expanded: I use `/define` to bind them in a temporary environment.

do-case (internal)	
keyexpr	an expression
clauses	a Lisp list of expressions
env	an environment
Returns:	an expression

```
713 proc ::constcl::do-case {keyexpr clauses env} {
714   if {[T [null? $clauses]]} {
715     return [parse "'()"]
716   } else {
```

If the length of the *clauses* is greater than 0, extract a *datumlist* and a *body* from the first clause. Then build a *predicate* of the form `(memv keyexpr (quote datumlist))`.

```
717     set datumlist [caar $clauses]
718     set body [cdar $clauses]
719     set predicate [list [S memv] $keyexpr \
720         [list [S quote] $datumlist]]
```

If the length of the *clauses* is 1, meaning that this is the last clause, and an `else` is found instead of a datumlist, set the predicate to `#t`.

```
721     if {[T [eq? [length $clauses] [N 1]]]} {
722       if {[T [eq? [caar $clauses] [S else]]]} {
723         set predicate ${::#t}
724       }
725     }
```

Finally, build a quasiquote structure and expand it to get the expansion of the `case` expression.

```
726     set env [MkEnv $env]
727     /define [S predicate] $predicate $env
728     /define [S body] $body $env
729     /define [S rest] [
730       do-case $keyexpr [cdr $clauses] $env] $env
731     set qq "`(if ,predicate
732                (begin ,@body)
733                ,rest)"
734     set expr [expand-quasiquote [parse $qq] $env]
735     $env destroy
736     return $expr
737   }
738 }
```

cond special form

`cond` is the third conditional form. The `cond` form has a list of clauses, each with a predicate and a body. The clauses is considered in order, and if a predicate evaluates to something other than `#f` the body is evaluated and the remaining clauses are ignored.

Syntax: (**cond** *clause* ...)

where each *clause* has the form

(*test expression* ...)

or

(*test* **=>** *recipient*)

where *recipient* is a procedure that accepts one argument, which is evaluated with the result of the predicate as argument if the predicate returns a true value.

The last clause may have the form

(**else** *expression* ...)

Example:

```
(let ((a 3))
  (cond ((> a 3) 'greater)
        ((< a 3) 'less)
        (else 'equal)))      ==> equal
```

The `cond` special form is expanded by `special-cond`. It expands to `'()` if there are no clauses (left), and to nested `if` constructs if there are some.

special-cond procedure

special-cond (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
reg special cond

proc ::constcl::special-cond {expr env} {
  set expr [do-cond [cdr $expr] $env]
  eval $expr $env
}
```

do-cond procedure

`do-cond` is called recursively for every clause of the `cond` form. It chops up the clause into predicate and body. In the last clause, the predicate is allowed to be `else` (which gets translated to `#t`). If there is no body, the body is set to the predicate. The form is expanded to a recursive `if` form.

do-cond (internal)	
tail	a Lisp list of expressions
env	an environment
Returns:	an expression

```
proc ::constcl::do-cond {tail env} {
  set clauses $tail
  if {[T [null? $clauses]]} {
    return [parse "'()"]
  } else {
```

If the length of the *clauses* is greater than 0, extract a *predicate* and a *body* from the first clause.

```
750     set predicate [caar $clauses]
751     set body [cdar $clauses]
```

If the length of the *clauses* is 1, meaning that this is the last clause, and an `else` is found instead of a predicate, set the predicate to `#t`.

```
752     if {[T [eq? [length $clauses] [N 1]]]} {
753       if {[T [eq? $predicate [S else]]]} {
754         set predicate ${::#t}
755       }
756     }
```

If there is a `=>` between the *predicate* and the *body*, rewrite the *body* to call the `caddar` of the *clauses* with the result of *predicate* as argument.

```
757     if {[T [symbol? [car $body]]] &&
758         [[car $body] name] eq "=>"} {
759       set body [list [caddar $clauses] $predicate]
```

Otherwise, if the *body* is empty, set *body* to *predicate*. If *body* has one or more expressions, wrap them in `begin`.

```
760     } else {
761       if {[[length $body] numval] == 0} {
762         set body $predicate
763       } else {
764         set body [cons [S begin] $body]
765       }
766     }
```

Finally, build a quasiquote structure and expand it to get the expansion of the cond expression.

```
767     set env [MkEnv $env]
768     /define [S predicate] $predicate $env
769     /define [S body] $body $env
770     /define [S rest] [
771       do-cond [cdr $clauses] $env] $env
772     set qq "`(if ,predicate
773                  ,body
774                  ,rest)"
775     set expr [expand-quasiquote [parse $qq] $env]
776     $env destroy
777     return $expr
778   }
779 }
```

3.5 Sequence

Example: `(begin (define r 10) (* r r))` $\Rightarrow$ *100*

There are times when one wants to treat a number of expressions as if they were one single expression (e.g. in the consequent or alternate of an `if` form). The `begin` special form bundles up expressions as an aggregate form. Internally, it sees to it that all the expressions are evaluated in order and that the resulting value of the last one is returned as the aggregate's result.

As part of the processing of sequences, *local defines* are resolved (see page 116), acting on expressions of the form `(begin (define ...` when in a local environment.

The following forms have an implicit `begin` in their bodies and the use of `begin` is therefore unnecessary with them:

`case`, `cond`, `define` (procedure define only), `lambda`, `let`, `let*`, `letrec`.

begin special form

Syntax: (**begin** *expression* ...)

The `begin` special form is expanded by `special-begin`.

special-begin (internal)	
expr	an expression
env	an environment
Returns:	a value

```
reg special begin

proc ::constcl::special-begin {expr env} {
  if {$env ne "::constcl::global_env" &&
    [T [pair? [cadr $expr]]] &&
    [T [eq? [caadr $expr] [S define]]]
  } then {
    set expr [resolve-local-defines $expr]
    eval $expr $env
  } else {
    /begin [cdr $expr] $env
  }
}
```

/begin procedure

The `/begin` helper procedure takes a Lisp list of expressions and evaluates them in sequence, returning the value of the last one.

/begin (internal)	
exps	a Lisp list of expressions
env	an environment
Returns:	a value

```
793 proc ::constcl::/begin {exps env} {
794   if {[T [pair? $exps]]} {
795     if {[T [pair? [cdr $exps]]]} {
796       eval [car $exps] $env
797       return [/begin [cdr $exps] $env]
798     } else {
799       return [eval [car $exps] $env]
800     }
801   } else {
802     return [parse "'()"]
803   }
804 }
```

3.6 Definition

Example: `(define r 10)` ⇒ *...(a definition doesn't evaluate to anything)*

We've already seen the relationship between symbols and values. Through (variable) definition, a symbol is bound to a value (or rather to the location the value is in), creating a variable. The `/define` helper procedure adds a variable to the current environment. It first checks that the symbol name is a valid identifier and that it isn't already bound in the current environment. Then it updates the environment with the new binding.

The syntaxes with `lambda` in them refer to the eight syntactic form, procedure definition (see page 80).

define special form

Syntax: either

(**define** *variable expression*)

(**define** (*variable formals*) *body*)

where *formals* is a proper or dotted list of identifiers; equivalent form:

(**define** *variable* (**lambda** (*formals*) *body*)).

or

(**define** (*variable* . *formal*) *body*)

where *formal* is a single identifier; equivalent form:

(**define** *variable* (**lambda** *formal body*))

body should be one or more expressions.

The `define` special form is expanded by `special-define`.

special-define (internal)	
expr	an expression
env	an environment
Returns:	nothing

```
reg special define

proc ::constcl::special-define {expr env} {
  set expr [rewrite-define $expr $env]
  set sym [cadr $expr]
  set val [eval [caddr $expr] $env]
  /define $sym $val $env
}
```

rewrite-define procedure

`rewrite-define` rewrites "procedural define" syntaxes to their equivalent forms with `lambda`, which unifies the syntaxes with (**define** *variable expression*). That syntax passes through `rewrite-define` unchanged.

rewrite-define (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
813 proc ::constcl::rewrite-define {expr env} {
814   if {[T [pair? [cadr $expr]]]} {
815     set tail [cdr $expr]
816     set env [::constcl::MkEnv $env]
817     /define [S tail] $tail $env
818     set qq "`(define ,(caar tail)
819                 (lambda ,(cdar tail) ,@(cdr tail)))"
820     set expr [expand-quasiquote [parse $qq] $env]
821     $env destroy
822   }
823   return $expr
824 }
```

/define procedure

The `/define` helper procedure carries out the binding of a symbol in a given environment, and stores the value in the location of binding.

/define (internal)	
sym	a symbol
val	a value
env	an environment
Returns:	nothing

```
825 proc ::constcl::/define {sym val env} {
826   varcheck [idcheck [$sym name]]
827   # will throw an error if $sym is bound
828   $env bind $sym VARIABLE $val
829   return
830 }
```

3.7 Assignment

Example: `(set! r 20)` ⇒ *20 (*`r` *is a bound symbol, so it's allowed to assign to it)*

Once again we consider the relationship of a symbol, an environment, and a value. Once a symbol is bound to a location in the environment, the value at that location can be changed with reference to the symbol, altering the value of the variable. The process is called assignment.

It is carried out by the `set!` special form. Given a symbol and a value, it finds the symbol's binding environment and updates the location with the value. It returns the value, so calls to `set!` can be chained: `(set! foo (set! bar 99))` sets both variables to 99. By Scheme convention, procedures that modify variables have '!' at the end of their name.

It is an error to do assignment on an unbound symbol.

set! special form

Syntax: (**set!** *variable expression*)

The `set!` special form is expanded by `special-set!`.

special-set! (internal)	
expr	an expression
env	an environment
Returns:	a value

```
831 reg special set!
832
833 proc ::constcl::special-set! {expr env} {
834   set args [cdr $expr]
835   set var [car $args]
836   set val [eval [cadr $args] $env]
837   [$env find $var] assign $var VARIABLE $val
838   set val
839 }
```

3.8 Procedure definition

Example: `(lambda (r) (* r r))` ⇒ `::oo::Obj3601` *(it will be a different object each time)*

In Lisp, procedures are values just like numbers or characters. They can be defined as the value of a variable, passed to other procedures, and returned from procedures. One difference from most values is that procedures need to be specified. Two questions must answered: what is the procedure meant to do? The code that does that will form the *body* of the procedure. Also, which, if any, items of data (*parameters*) will have

to be provided to the procedure to make it possible to calculate its result?

As an example, imagine that we want to have a procedure that calculates the square (`x * x`) of a given number. In Lisp, expressions are written with the operator first and then the operands: `(* x x)`. That is the body of the procedure. Now, what data will we have to provide to the procedure to make it work? A value stored in the variable `x` will do. It's only a single parameter, but by custom we need to put it in a list: `(x)`. The operator that creates procedures is called `lambda`, and we create the function with `(lambda (x) (* x x))`.

One more step is needed before we can use the procedure. It must have a name. We could define it like this: `(define square (lambda (x) (* x x)))` but there is actually a shortcut notation for it: `(define (square x) (* x x))`.

Now, `square` is pretty tame. How about the `hypotenuse` procedure? `(define (hypotenuse a b) (sqrt (+ (square a) (square b))))`. It calculates the square root of the sum of two squares.

The `lambda` special form makes a Procedure object (see page 201). First it needs to wrap `body` inside a `begin` (`S begin` stands for 'the symbol begin'). The Lisp list `formals` (for *formal parameters*) is passed on as it is.

lambda special form

Syntax: (**lambda** *formals body*)

where *body* is one or more expressions.

The `lambda` special form is expanded by `special-lambda`.

Scheme formal parameters lists

A Scheme formals list is either:

- An *empty list*, `()`, meaning that no arguments are accepted,
- A *proper list*, `(a b c)`, meaning it accepts three arguments, one in each symbol,
- A *symbol*, `a`, meaning that all arguments go into `a`, or
- A *dotted list*, `(a b . c)`, meaning that two arguments go into `a` and `b`, and the rest into `c`.

special-lambda (internal)	
expr	an expression
env	an environment
Returns:	a procedure

```
reg special lambda

proc ::constcl::special-lambda {expr env} {
  set args [cdr $expr]
  set formals [car $args]
  set body [cons [S begin] [cdr $args]]
  return [MkProcedure $formals $body $env]
}
```

3.9 Procedure call

Example: `(+ 1 6)` $\Rightarrow$ *7*

Once we have procedures, we can *call* them to have their calculations performed and yield results. The procedure name is put in the operator position at the front of a list, and the operands follow in the rest of the list. Our `square` procedure would be called for instance like this: `(square 11)`, and it would return 121.

`invoke` arranges for a procedure to be called with each of the values in the *argument list* (the list of operands). It checks if *pr* really is a procedure, and determines whether to call *pr* as a `call` method invocation or as a Tcl command. Before `invoke` is called, the argument list should be evaluated with `eval-list` (see page 98).

invoke procedure

invoke (internal)	
pr	a procedure
vals	a Lisp list of values
Returns:	what pr returns

```
848 proc ::constcl::invoke {pr vals} {
849   check {procedure? $pr} {
850     PROCEDURE expected\n([$pr tstr] val ...)
851   }
852   if {[info object isa object $pr]} {
853     $pr call {*}[splitlist $vals]
854   } else {
855     $pr {*}[splitlist $vals]
```

```
856     }
857 }
```

3.10 Binding forms

The binding forms are not fundamental the way the earlier nine forms are. They are an application of a combination of forms eight and nine, the procedure definition form and the procedure call form. But their use is sufficiently distinguished to earn them their own heading.

let special form

Syntax: (**let** ((*variable init*) ...) *body*)
or ("named let")
(**let** *variable* ((*variable init*) ...) *body*)
where *body* is one or more expressions.

The `let` special form (both forms) is expanded by `special-let`. They are ultimately rewritten to calls to `lambda` constructs and evaluated as such.

special-let (internal)	
expr	an expression
env	an environment
Returns:	a value

```
858 reg special let
859
860 proc ::constcl::special-let {expr env} {
861   if {[T [symbol? [cadr $expr]]]} {
```

```
862       set expr [rewrite-named-let $expr $env]
863     }
864     set expr [rewrite-let $expr $env]
865     eval $expr $env
866   }
```

rewrite-named-let procedure

rewrite-named-let (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
867 proc ::constcl::rewrite-named-let {expr env} {
```

The rewriter for named `let` chops up the expression into *variable*, *bindings*, and *body*.

```
868     set variable [cadr $expr]
869     set bindings [caddr $expr]
870     set body [cdddr $expr]
```

It creates a dictionary with the *variable* as key and `#f` as value. Then it fills up the dictionary with variable/value pairs from the *bindings*.

```
871     set vars [dict create $variable ${::#f}]
872     parse-bindings vars $bindings
```

It uses the dictionary to build a declaration list for a `let` form, a variable list for a `lambda` form, and a procedure call.

Then it assembles a `let` form with the declaration list and a body consisting of an assignment and the procedure call. The assignment binds the variable to a `lambda` form with the varlist and the original *body*. The `let` form is returned, meaning that the primary expansion of the named `let` is a regular `let` form.

```
873   set env [MkEnv $env]
874   /define [S decl] [list {*}[dict values [
875     dict map {k v} $vars {list $k $v}]]] $env
876   /define [S variable] $variable $env
877   /define [S varlist] [list {*}[lrange [
878     dict keys $vars] 1 end]] $env
879   /define [S body] $body $env
880   /define [S call] [list {*}[
881     dict keys $vars]] $env
882   set qq "`(let ,decl
883               (set!
884                 ,variable
885                   (lambda ,varlist ,@body)) ,call)"
886   set expr [expand-quasiquote [parse $qq] $env]
887   $env destroy
888   return $expr
889 }
```

rewrite-let procedure

rewrite-let (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
890 proc ::constcl::rewrite-let {expr env} {
```

The rewriter for regular `let` chops up the original expression into *bindings* and *body*.

```
891   set bindings [cadr $expr]
892   set body [cddr $expr]
```

It creates an empty dictionary and fills it up with variable/-value pairs from the *bindings*.

```
893   set vars [dict create]
894   parse-bindings vars $bindings
```

Then it builds a `lambda` operator form with the variable list, the *body*, and the value list. The `lambda` call is returned as the expansion of the regular `let` form.

```
895   set env [MkEnv $env]
896   /define [S varlist] [list {*}[
897     dict keys $vars]] $env
898   /define [S body] $body $env
899   /define [S vallist] [list {*}[
900     dict values $vars]] $env
901   set qq "`((lambda ,varlist ,@body)
902               ,@vallist)"
903   set expr [expand-quasiquote [parse $qq] $env]
904   $env destroy
905   return $expr
906 }
```

parse-bindings procedure

`parse-bindings` is a helper procedure that traverses a `let` bindings list and extracts variables and values, which it puts in

a dictionary. It throws an error if a variable occurs more than once.

parse-bindings (internal)	
name	a call-by-name name
bindings	a Lisp list of values
Returns:	nothing

```
907 proc ::constcl::parse-bindings {name bindings} {
908   upvar $name vars
909   foreach binding [splitlist $bindings] {
910     set var [car $binding]
911     set val [cadr $binding]
912     if {$var in [dict keys $vars]} {
913         ::error "'[$var name]' occurs more than once"
914     }
915     dict set vars $var $val
916   }
917   return
918 }
```

letrec special form

The `letrec` form is similar to `let`, but the bindings are created before the values for them are calculated. This means that one can define mutually recursive procedures.

Syntax: (**letrec** ((*variable init*) ...) *body*)

where *body* is one or more expressions.

The `letrec` special form is expanded by `special-letrec`.

special-letrec (internal)	
expr	an expression
env	an environment
Returns:	a value

```
reg special letrec

proc ::constcl::special-letrec {expr env} {
  set expr [rewrite-letrec $expr $env]
  eval $expr $env
}
```

rewrite-letrec procedure

rewrite-letrec (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
proc ::constcl::rewrite-letrec {expr env} {
```

The rewriter for `letrec` chops up the original expression into *bindings* and *body*.

```
  set bindings [cadr $expr]
  set body [cddr $expr]
```

It creates an empty dictionary and fills it up with variable/-value pairs from the *bindings*.

```
  set vars [dict create]
  parse-bindings vars $bindings
```

The keys and values in the dictionary are used to create three dictionaries: one for the outer lambda, one for the inner lambda, and one for the assignments.

```
930   foreach {key val} $vars {
931     dict set outer $key [list [S quote] ${::#UND}]
932     dict set inner [set g [gensym "g"]] $val
933     dict set assigns $key $g
934   }
```

The three dictionaries are used to populate a double lambda construct in a quasiquote structure, which is expanded and returned.

```
935   set env [MkEnv $env]
936   # outer vars
937   /define [S ovars] [
938     list {*}[dict keys $outer]] $env
939   # outer vals
940   /define [S ovals] [
941     list {*}[dict values $outer]] $env
942   # inner vars
943   /define [S ivars] [
944     list {*}[dict keys $inner]] $env
945   # inner vals
946   /define [S ivals] [
947     list {*}[dict values $inner]] $env
948   /define [S assigns] [list {*}[lmap {k v} $assigns {
949       list [S set!] $k $v
950     }]] $env
951   /define [S body] $body $env
952   set qq "‘((lambda ,ovars
953               ((lambda ,ivars ,@assigns) ,@ivals)
954               ,@body) ,@ovals)"
```

```
955   set expr [expand-quasiquote [parse $qq] $env]
956   $env destroy
957   return $expr
958 }
```

let* special form

The `let*` form is similar to `let`, but the items in the binding list are considered sequentially, so the initializer in the second or later binding can reference the first binding, etc.

Syntax: (**let*** ((*variable init*) ...) *body*)

where *body* is one or more expressions.

The `let*` special form is expanded by `special-let*`.

special-let* (internal)	
expr	an expression
env	an environment
Returns:	a value

```
959 reg special let*
960
961 proc ::constcl::special-let* {expr env} {
962   set expr [rewrite-let* [cadr $expr] [cddr $expr] $env]
963   eval $expr $env
964 }
```

rewrite-let* procedure

rewrite-let* (internal)	
bindings	a Lisp list of values
body	a Lisp list of expressions
env	an environment
Returns:	an expression

```
965 proc ::constcl::rewrite-let* {bindings body env} {
966   set env [MkEnv $env]
967   if {$bindings eq ${::#NIL}} {
```

If there are no more bindings, wrap the *body* in a `begin` and return it.

```
968     /define [S body] $body $env
969     set qq "`(begin ,@body)"
970     set expr [expand-quasiquote [parse $qq] $env]
971   } else {
```

Otherwise, create a quasiquote structure with a lambda call and put a variable and a value at a time in it. The body of the lambda is the rewriter itself called recursively.

Return the lambda call.

```
972     /define [S var] [caar $bindings] $env
973     /define [S val] [cadar $bindings] $env
974     /define [S rest] [rewrite-let* [cdr $bindings] \
975       $body $env] $env
976     set qq "`((lambda (,var)
977                 ,rest) ,val)"
978     set expr [expand-quasiquote [parse $qq] $env]
979   }
980   $env destroy
```

```
981     return $expr
982 }
```

3.11 Environments

Before I can talk about the evaluator, I need to spend some time on environments. To simplify, an environment can be seen as a table–or spreadsheet, if you will–that connects (binds) names to cells, which contain values. The evaluator looks up values in the environment that way. But there's more to an environment than just a name-value coupling. The environment also contains references to the procedures that make up the Lisp library. And their bindings aren't just a simple connection: there are several kinds of bindings, from variable binding, the most common one, to special-form bindings for the fundamental operations of the interpreter, and syntax bindings for the macros that get expanded to 'normal' code.

There isn't just one environment, either. Every time a non-primitive procedure is called, a new environment is created, one which has bindings for the procedure formal parameters and which links to the environment that was current when the procedure was defined (which in turn links backwards all the way to the original global environment). The evaluator follows into the new environment to evaluate the body of the procedure there, and then as the evaluator goes back along the call stack, it sheds environment references.

Not only procedures but binding forms (such as `let`) create new environments for the evaluator to work in. As they

do that, they also bind variables to values. Just like with procedures, the added local bindings can shadow bindings in underlying environments but does not affect them: once the local environment has been forgotten by the evaluator, the underlying bindings are once more visible. The other side of the coin is that temporary environments don't have to be complete: every binding that the evaluator can't find in a temporary environment it looks for in the parent environment, or its parent and so on.

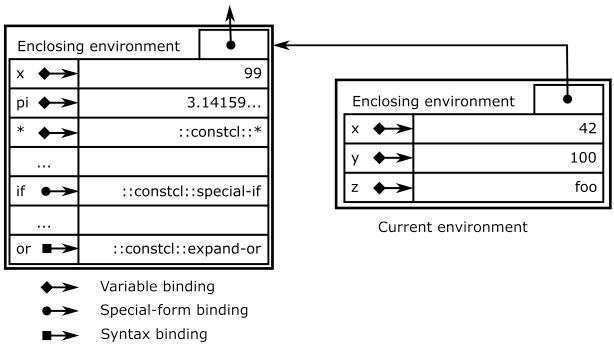
Environments make up the world the evaluator lives in and are the source of its values and procedures. The ability of procedure calls and execution of binding forms to temporarily change the current environment is a powerful one.

From the evaluator's perspective it uses the environment to keep track of changes in the state of the evaluation. In this way, the evaluator uses the environment for continuity and a progress record.

I will talk some more about the implementation of environments in a later section.

3.12 The evaluator

Now that all nine syntactic forms are in place and we have a basic understanding of the environment, we can start assembling the evaluator.



Two sample environments

eval procedure

The heart of the Lisp interpreter, `eval` takes a Lisp expression and processes it according to its form. Variable reference and constant literals are handled here, but the other seven syntactic forms are delegated to `eval-form`.

eval (public)	
expr	an expression
?env?	an environment
Returns:	a value

```
983 reg eval
984
985 proc ::constcl::eval \
986   {expr {env ::constcl::global_env}} {
987   if {[T [symbol? $expr]]} {
```

```
988       lookup $expr $env
989     } elseif {[T [self-evaluating? $expr]]} {
990       set expr
991     } elseif {[T [pair? $expr]]} {
992       eval-form $expr $env
993     } else {
994       error "unknown expression type [$expr tstr]"
995     }
996   }
```

eval-form procedure

If the `car` of the expression (the operator) is a symbol, `eval-form` looks at the *binding information* (which the `reg` procedure (see page 3) puts into the standard library and thereby the global environment) for the symbol. The *binding type* tells in general how the expression should be treated: as a special form, a variable, or a macro (see page 99). The *handling info* gives the exact procedure that will take care of the expression. If the operator isn't a symbol, it is evaluated and applied to the evaluated rest of the expression.

The seven remaining syntactic forms (and the binding forms) are implemented as one or more special forms and handled when the relevant symbol appears in the `car` of the expression. Their *binding type* is `SPECIAL` and the *handling info* consists of the name of the procedure expanding the special form. The procedure is called with the expression and the environment as arguments.

eval-form (internal)	
expr	an expression
env	an environment
Returns:	a value

```
997  proc ::constcl::eval-form {expr env} {
998    set op [car $expr]
999    set args [cdr $expr]
1000   if {[T [symbol? $op]]} {
1001     lassign [binding-info $op $env] btype hinfo
1002     switch $btype {
1003       UNBOUND {
1004         error "unbound symbol" $op
1005       }
1006       SPECIAL {
1007         $hinfo $expr $env
1008       }
1009       VARIABLE {
1010         invoke $hinfo [eval-list $args $env]
1011       }
1012       SYNTAX {
1013         eval [$hinfo $expr $env] $env
1014       }
1015       default {
1016         error "unrecognized binding type" $btype
1017       }
1018     }
1019   } else {
1020     invoke [eval $op $env] [eval-list $args $env]
1021   }
1022 }
```

binding-info procedure

The `binding-info` procedure takes a symbol and returns a list of two items: 1) the binding type of the symbol, and 2) the handling info that `eval-form` uses to handle this symbol.

binding-info (internal)	
op	a symbol
env	an environment
Returns:	binding info

```
1023 proc ::constcl::binding-info {op env} {
1024   set actual_env [$env find $op]
1025   if {$actual_env eq "::constcl::null_env"} {
1026     return [::list UNBOUND {}]
1027   } else {
1028     return [$actual_env get $op]
1029   }
1030 }
```

eval-list procedure

`eval-list` successively evaluates the elements of a Lisp list and returns the collected results as a Lisp list.

eval-list (internal)	
exps	a Lisp list of expressions
env	an environment
Returns:	a Lisp list of values

```
1031 proc ::constcl::eval-list {exps env} {
1032   if {[T [pair? $exps]]} {
1033     return [cons [eval [car $exps] $env] \
1034       [eval-list [cdr $exps] $env]]
1035   } else {
1036     return ${::#NIL}
```

```
1037    }
1038 }
```

3.13 Macros

One of Lisp's strong points is macros that allow concise, abstract expressions that are automatically rewritten into other, more concrete but also more verbose expressions. This interpreter does macro expansion, but the user can't define new macros–the ones available are hardcoded in the code below.

A macro expander procedure takes an expression and an environment as parameters. In the end, the expanded expression is passed back to `eval`.

expand-and procedure

`expand-and` expands the `and` macro. It returns a `begin`-expression if the macro has 0 or 1 elements, and a nested `if` construct otherwise.

expand-and (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
1039 reg macro and
1040
1041 proc ::constcl::expand-and {expr env} {
1042   set tail [cdr $expr]
1043   if {[[length $tail] numval] == 0} {
```

```
    list [S begin] ${::#t}
  } elseif {[[length $tail] numval] == 1} {
    cons [S begin] $tail
  } else {
    do-and $tail ${::#t} $env
  }
}
```

do-and procedure

`do-and` is called recursively for every argument of `expand-and` if there is more than one.

do-and (internal)	
tail	an expression tail
prev	an expression
env	an environment
Returns:	an expression

```
proc ::constcl::do-and {tail prev env} {
  if {[T [null? $tail]]} {
    return $prev
  } else {
    set env [MkEnv $env]
    /define [S first] [car $tail] $env
    /define [S rest] [do-and [cdr $tail] \
        [car $tail] $env] $env
    set qq "`(if ,first ,rest #f)"
    set expr [expand-quasiquote [parse $qq] $env]
    $env destroy
    return $expr
  }
}
```

expand-del! procedure

The macro `del!` updates a property list. It removes a key-value pair if the key is present, or leaves the list untouched if it isn't.

expand-del! (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
1065 reg macro del!
1066
1067 proc ::constcl::expand-del! {expr env} {
1068   set tail [cdr $expr]
1069   set env [MkEnv $env]
1070   if {[T [null? $tail]]} {
1071     ::error "too few arguments, 0 of 2"
1072   }
1073   /define [S listname] [car $tail] $env
1074   if {[T [null? [cdr $tail]]]} {
1075     ::error "too few arguments, 1 of 2"
1076   }
1077   /define [S key] [cadr $tail] $env
1078   set qq "`(set! ,listname
1079               (delete! ,listname ,key))"
1080   set expr [expand-quasiquote [parse $qq] $env]
1081   $env destroy
1082   return $expr
1083 }
```

expand-for procedure

The `expand-for` procedure expands the `for` macro. It returns a `begin` construct containing the iterations of each clause (multiple clauses weren't implemented for the longest time, but I brought up my strongest brain cells and they did it).

expand-for (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
1084 reg macro for
1085
1086 proc ::constcl::expand-for {expr env} {
1087   set res [do-for [cdr $expr] $env]
1088   lappend res [parse "'()"]
1089   return [list [S begin] {*}$res]
1090 }
```

for-seq procedure

`for-seq` is a helper procedure that sets up the sequence of values that the iteration is based on. First it evaluates the code that generates the sequence, and then it converts it to a Tcl list.

for-seq (internal)	
seq	an expression
env	an environment
Returns:	a Tcl list of values

```
1091 proc ::constcl::for-seq {seq env} {
```

If *seq* is a number, call the `in-range` procedure to get a sequence. Otherwise, evaluate *seq*.

```
1092    if {[T [number? $seq]]} {
1093      set seq [in-range $seq]
1094    } else {
1095      set seq [eval $seq $env]
1096    }
```

Make the sequence a Tcl list, one way or another.

```
1097    if {[T [list? $seq]]} {
1098      set seq [splitlist $seq]
1099    } elseif {[T [string? $seq]]} {
1100      set seq [lmap c [split [$seq value] {}] {
1101        switch $c {
1102          " "  { MkChar #\\space }
1103          "\n" { MkChar #\\newline }
1104          default {
1105            MkChar #\\$c
1106          }
1107        }
1108      }]
1109    } elseif {[T [vector? $seq]]} {
1110      set seq [$seq value]
1111    } else {
1112      ::error "unknown sequence type [$seq tstr]"
1113    }
1114    return $seq
1115  }
```

do-for procedure

`do-for` is another helper procedure which does most of the work in the `for/*` forms. It iterates over the clauses, extracting

and preparing the sequence for each, and stores each of the sequence steps in a dictionary under a double key: the identifier and the ordinal of the step.

Then it creates a `let` construct for each step, in which each of the clauses' identifiers is bound to the step's value. The Tcl list of `let` constructs is returned.

Each clause's sequence is supposed to be the same length as the others. One weakness of this implementation is that it doesn't ensure this, just hopes that the user does the right thing.

do-for (internal)	
tail	an expression tail
env	an environment
Returns:	a Tcl list of expressions

```
1116 proc ::constcl::do-for {tail env} {
1117   # make clauses a Tcl list
1118   set clauses [splitlist [car $tail]]
1119   set body [cdr $tail]
1120   set data [dict create]
1121   set length 0
1122   foreach clause $clauses {
1123     set id [car $clause]
1124     set sequence [for-seq [cadr $clause] $env]
1125     set length [llength $sequence]
1126     # save every id and step of the iteration
1127     for {set i 0} {$i < $length} {incr i} {
1128         dict set data $id $i [lindex $sequence $i]
1129     }
1130   }
1131   set res {}
1132   # for every step of the iteration...
1133   for {set i 0} {$i < $length} {incr i} {
1134     set decl {}
```

```
1135      # retrieve the ids
1136      foreach id [dict keys $data] {
1137        # list the id and the step
1138        lappend decl [
1139          list $id [dict get $data $id $i]]
1140      }
1141      # add to the structure of let constructs
1142      lappend res [list [S let] [
1143          list {*}$decl] {*}[splitlist $body]]
1144    }
1145    return $res
1146  }
```

expand-for/and procedure

The `expand-for/and` procedure expands the `for/and` macro. It returns an `and` construct containing the iterations of the clauses.

The only differences from `expand-for` is that it doesn't add `(quote ())` and that it wraps the list of iterations in `and` instead of `begin`.

expand-for/and (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
1147  reg macro for/and
1148
1149  proc ::constcl::expand-for/and {expr env} {
1150    set tail [cdr $expr]
1151    set res [do-for $tail $env]
1152    return [list [S and] {*}$res]
```

```
1153 }
```

expand-for/list procedure

The `expand-for/list` procedure expands the `for/list` macro. It returns a `list` construct containing the iterations of each clause.

The only difference from `expand-for/and` is that it wraps the list of iterations in `list` instead of `and`.

expand for/list (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
1154 reg macro for/list
1155
1156 proc ::constcl::expand-for/list {expr env} {
1157   set tail [cdr $expr]
1158   set res [do-for $tail $env]
1159   return [list [S list] {*}$res]
1160 }
```

expand-for/or procedure

The `expand-for/or` procedure expands the `for/or` macro. It returns an `or` construct containing the iterations of each clause.

The only difference from `expand-for/list` is that it wraps the list of iterations in `or` instead of `list`.

expand-for/or (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
reg macro for/or

proc ::constcl::expand-for/or {expr env} {
  set tail [cdr $expr]
  set res [do-for $tail $env]
  return [list [S or] {*}$res]
}
```

expand-or procedure

`expand-or` expands the `or` macro. It returns a `begin`-expression if the macro has 0 or 1 elements, and a nested `if` construct otherwise.

expand-or (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
reg macro or

proc ::constcl::expand-or {expr env} {
  set tail [cdr $expr]
  if {[[length $tail] numval] == 0} {
    return [list [S begin] ${::#f}]
  } elseif {[[length $tail] numval] == 1} {
    return [cons [S begin] $tail]
```

```
1176    } else {
1177      return [do-or $tail $env]
1178    }
1179  }
```

do-or procedure

`do-or` is called recursively for each argument to `expand-or` if there is more than one argument.

do-or (internal)	
tail	an expression tail
env	an environment
Returns:	an expression

```
1180  proc ::constcl::do-or {tail env} {
1181    if {[T [null? $tail]]} {
1182      return ${::#f}
1183    } else {
1184      set env [MkEnv $env]
1185      /define [S first] [car $tail] $env
1186      /define [S rest] [do-or [cdr $tail] $env] $env
1187      set qq "`(let ((x ,first)) (if x x ,rest))"
1188      set expr [expand-quasiquote [parse $qq] $env]
1189      $env destroy
1190      return $expr
1191    }
1192  }
```

expand-pop! procedure

The macro `pop!` updates a list. It removes the first element.

expand-pop! (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
reg macro pop!

proc ::constcl::expand-pop! {expr env} {
  set tail [cdr $expr]
  set env [MkEnv $env]
  if {[T [null? $tail]]} {
      ::error "too few arguments:\n(pop! listname)"
  }
  if {[symbol? [car $tail]] eq ${::#f}} {
      ::error "SYMBOL expected:\n(pop! listname)"
  }
  /define [S listname] [car $tail] $env
  set qq "`(set! ,listname (cdr ,listname))"
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

expand-push! procedure

The macro `push!` updates a list. It adds a new element as the new first element. The `push!` and `pop!` macros together implement a stack on a list.

expand-push! (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
reg macro push!

proc ::constcl::expand-push! {expr env} {
  set tail [cdr $expr]
  set env [MkEnv $env]
  if {[T [null? $tail]]} {
    ::error \
      "too few arguments:\n(push! obj listname)"
  }
  /define [S obj] [car $tail] $env
  if {[T [null? [cdr $tail]]]} {
    ::error \
      "too few arguments:\n(push! obj listname)"
  }
  if {[symbol? [cadr $tail]] eq ${::#f}} {
    ::error \
      "SYMBOL expected:\n(push! obj listname)"
  }
  /define [S listname] [cadr $tail] $env
  set qq "`(set!
             ,listname
             (cons ,obj ,listname))"
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

expand-put! procedure

The macro `put!` updates a property list. It adds a key-value pair if the key isn't present, or changes the value in place if it is.

expand-put! (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
reg macro put!

proc ::constcl::expand-put! {expr env} {
  set tail [cdr $expr]
  set env [::constcl::MkEnv $env]
  if {[T [null? $tail]]} {
      ::error "too few arguments, 0 of 3"
  }
  /define [S name] [car $tail] $env
  if {[T [null? [cdr $tail]]]} {
      ::error "too few arguments, 1 of 3"
  }
  /define [S key] [cadr $tail] $env
  if {[T [null? [cddr $tail]]]} {
      ::error "too few arguments, 2 of 3"
  }
  /define [S val] [caddr $tail] $env
  set qq "`(let ((idx (list-find-key ,name ,key)))
             (if (< idx 0)
               (set!
                 ,name
                 (append (list ,key ,val) ,name))
               (begin
                 (list-set! ,name (+ idx 1) ,val)
                 ,name)))"
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

expand-quasiquote procedure

A quasi-quote isn't a macro, but we will deal with it in this section anyway. `expand-quasiquote` traverses a quasi-quoted structure searching for `unquote` and `unquote-splicing`. This code is brittle and sprawling and I barely understand it myself, but it works (and is the basis for a lot of the special form/macro expanders).

expand-quasiquote (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
1265 reg macro quasiquote
1266
1267 proc ::constcl::expand-quasiquote {expr env} {
1268   set tail [cdr $expr]
1269   set qqlevel 0
1270   if {[T [list? [car $tail]]]} {
1271     set node [car $tail]
1272     return [qq-visit-child $node 0 $env]
1273   } elseif {[T [vector? [car $tail]]]} {
1274     set vect [car $tail]
1275     set res {}
1276     for {set i 0} {$i < [
1277         [vector-length $vect] numval]} {incr i} {
1278       set idx [MkNumber $i]
1279       set vecref [vector-ref $vect $idx]
1280       if {[T [pair? $vecref]] &&
1281           [T [eq? [car $vecref] [
1282             S unquote]]]} {
1283         if {$qqlevel == 0} {
1284           lappend res [eval [cadr $vecref] $env]
1285         }
```

```
      } elseif {[T [pair? $vecref]] &&
          [T [eq? [car $vecref] [
            S unquote-splicing]]]} {
        if {$qqlevel == 0} {
          lappend res {*}[splitlist [
            eval [cadr $vecref] $env]]
        }
      } elseif {[T [atom? $vecref]]} {
        lappend res $vecref
      } else {
      }
    }
    return [list [S "vector"] {*}$res]
  }
}
```

qq-visit-child procedure

qq-visit-child (internal)	
node	a Lisp list of expressions
qqlevel	a Tcl number
env	an environment
Returns:	a Lisp list of expressions

```
proc ::constcl::qq-visit-child {node qqlevel env} {
  if {$qqlevel < 0} {
    set qqlevel 0
  }
  if {[T [list? $node]]} {
    set res {}
    foreach child [splitlist $node] {
      if {[T [pair? $child]] &&
          [T [eq? [car $child] [S unquote]]]} {
        if {$qqlevel == 0} {
          lappend res [eval [cadr $child] $env]
```

```
1312          } else {
1313            lappend res [list [S unquote] [
1314              qq-visit-child [cadr $child] [
1315              expr {$qqlevel - 1}] $env]]
1316          }
1317        } elseif {[T [pair? $child]] &&
1318            [T [eq? [car $child] [
1319            S unquote-splicing]]]} {
1320          if {$qqlevel == 0} {
1321            lappend res {*}[splitlist [
1322              eval [cadr $child] $env]]
1323          }
1324        } elseif {[T [pair? $child]] &&
1325            [T [eq? [car $child] [S quasiquote]]]} {
1326          lappend res [list [S quasiquote] [car [
1327            qq-visit-child [cdr $child] [
1328              expr {$qqlevel + 1}] $env]]
1329        } elseif {[T [atom? $child]]} {
1330          lappend res $child
1331        } else {
1332          lappend res [
1333            qq-visit-child $child $qqlevel $env]
1334        }
1335      }
1336    }
1337    return [list {*}$res]
1338  }
```

expand-unless procedure

`unless` is a conditional like `if`, but it takes a number of expressions. It executes them on a false outcome of `car $tail`.

expand-unless (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
1339 reg macro unless
1340
1341 proc ::constcl::expand-unless {expr env} {
1342   set tail [cdr $expr]
1343   set env [MkEnv $env]
1344   /define [S tail] $tail $env
1345   set qq "`(if ,(car tail)
1346              '()
1347              (begin ,@(cdr tail)))"
1348   set expr [expand-quasiquote [parse $qq] $env]
1349   $env destroy
1350   return $expr
1351 }
```

expand-when procedure

`when` is a conditional like `if`, but it takes a number of expressions. It executes them on a true outcome of `car $tail`.

expand-when (internal)	
expr	an expression
env	an environment
Returns:	an expression

```
1352 reg macro when
1353
1354 proc ::constcl::expand-when {expr env} {
```

```
1355    set tail [cdr $expr]
1356    set env [MkEnv $env]
1357    /define [S tail] $tail $env
1358    set qq "`(if ,(car tail)
1359                 (begin ,@(cdr tail))
1360                 '())"
1361    set expr [expand-quasiquote [parse $qq] $env]
1362    $env destroy
1363    return $expr
1364  }
```

3.14 Resolving local defines

This section is ported from 'Scheme 9 from Empty Space'. It rewrites local defines as a `letrec` form. `resolve-local-defines` takes a list of expressions and extracts variables and values from the defines in the beginning of the list. It builds a double lambda expression with the variables and values, and the rest of the expressions from the original list as body.

resolve-local-defines procedure

resolve-local-defines	
expr	an expression
Returns:	an expression

```
1365  proc ::constcl::resolve-local-defines {expr} {
1366    set exps [cdr $expr]
1367    set rest [lassign [
1368      extract-from-defines $exps VALS] a error]
```

```
1369    if {[T $error]} {
1370      return ${::#NIL}
1371    }
1372    set rest [lassign [
1373      extract-from-defines $exps VARS] v error]
1374    if {[T $error]} {
1375      return ${::#NIL}
1376    }
1377    if {$rest eq ${::#NIL}} {
1378      set rest [cons #UNS ${::#NIL}]
1379    }
1380    return [make-lambdas $v $a $rest]
1381  }
```

extract-from-defines procedure

`extract-from-defines` visits every define in the given list of expressions and extracts either a variable name or a value, depending on the state of the *part* flag, from each one of them. A Tcl list of 1) the resulting list of names or values, 2) error state, and 3) the rest of the expressions in the original list is returned.

extract-from-defines (internal)	
exps	a Lisp list of expressions
part	a flag, VARS or VALS
Returns:	a Tcl list of values

```
1382  proc ::constcl::extract-from-defines {exps part} {
1383    set a ${::#NIL}
1384    while {$exps ne ${::#NIL}} {
1385      if {[T [atom? $exps]] ||
1386          [T [atom? [car $exps]]] ||
```

```
1387          ![T [eq? [caar $exps] [S define]]]} {
1388        break
1389      }
1390      set n [car $exps]
1391      set k [length $n]
1392      if {![T [list? $n]] ||
1393          [$k numval] < 3 ||
1394          ![T [argument-list? [cadr $n]]] ||
1395          ([T [symbol? [cadr $n]]] &&
1396          [$k numval] > 3)} {
1397          return [::list ${::#NIL} ${::#t} ${::#NIL}]
1398        }
1399        if {[T [pair? [cadr $n]]]} {
1400          if {$part eq "VARS"} {
1401            set a [cons [caadr $n] $a]
1402          } else {
1403            set a [cons ${::#NIL} $a]
1404            set new [cons [cdadr $n] [cddr $n]]
1405            set new [cons [S lambda] $new]
1406            set-car! $a $new
1407          }
1408        } else {
1409          if {$part eq "VARS"} {
1410            set a [cons [cadr $n] $a]
1411          } else {
1412            set a [cons [caddr $n] $a]
1413          }
1414        }
1415        set exps [cdr $exps]
1416      }
1417      return [::list $a ${::#f} $exps]
1418  }
```

argument-list? procedure

`argument-list?` accepts a Scheme formals list and rejects other values.

argument-list? (internal)	
val	a value
Returns:	a boolean

```
1419 proc ::constcl::argument-list? {val} {
1420   if {$val eq ${::#NIL}} {
1421     return ${::#t}
1422   } elseif {[T [symbol? $val]]} {
1423     return ${::#t}
1424   } elseif {[T [atom? $val]]} {
1425     return ${::#f}
1426   }
1427   while {[T [pair? $val]]} {
1428     if {[symbol? [car $val]] eq ${::#f}} {
1429       return ${::#f}
1430     }
1431     set val [cdr $val]
1432   }
1433   if {$val eq ${::#NIL}} {
1434     return ${::#t}
1435   } elseif {[T [symbol? $val]]} {
1436     return ${::#t}
1437   }
1438 }
```

make-lambdas procedure

`make-lambdas` builds the `letrec` structure.

make-lambdas (internal)	
vars	a Lisp list of symbols
args	a Lisp list of expressions
body	a Lisp list of expressions
Returns:	an expression

```
1439 proc ::constcl::make-lambdas {vars args body} {
1440   set tmps [make-temporaries $vars]
1441   set body [append-b [
1442     make-assignments $vars $tmps] $body]
1443   set body [cons $body ${::#NIL}]
1444   set n [cons $tmps $body]
1445   set n [cons [S lambda] $n]
1446   set n [cons $n $args]
1447   set n [cons $n ${::#NIL}]
1448   set n [cons $vars $n]
1449   set n [cons [S lambda] $n]
1450   set n [cons $n [make-undefineds $vars]]
1451   return $n
1452 }
```

make-temporaries procedure

`make-temporaries` creates the symbols that will act as middlemen in transferring the values to the variables.

make-temporaries (internal)	
vals	a Lisp list of values
Returns:	a Lisp list of values

```
1453 proc ::constcl::make-temporaries {vals} {
1454   set res ${::#NIL}
```

```
1455    while {$vals ne ${::#NIL}} {
1456      set res [cons [gensym "g"] $res]
1457      set vals [cdr $vals]
1458    }
1459    return $res
1460  }
```

gensym procedure

gensym generates a unique symbol. The candidate symbol is compared to all the symbols in the symbol table to avoid collisions.

gensym (internal)	
prefix	a string
Returns:	a symbol

```
1461  proc ::constcl::gensym {prefix} {
1462    set symbolnames [
1463      dict keys $::constcl::symbolTable]
1464    set s $prefix<[incr ::constcl::gensymnum]>
1465    while {$s in $symbolnames} {
1466      set s $prefix<[incr ::constcl::gensymnum]>
1467    }
1468    return [S $s]
1469  }
```

append-b procedure

append-b joins two lists together.

append-b (internal)	
a	a Lisp list of values
b	a Lisp list of values
Returns:	a Lisp list of values

```
1470 proc ::constcl::append-b {a b} {
1471   if {$a eq ${::#NIL}} {
1472     return $b
1473   }
1474   set p $a
1475   while {$p ne ${::#NIL}} {
1476     if {[T [atom? $p]]} {
1477       ::error "append: improper list"
1478     }
1479     set last $p
1480     set p [cdr $p]
1481   }
1482   set-cdr! $last $b
1483   return $a
1484 }
```

make-assignments procedure

`make-assignments` creates the structure that holds the assignment statements. Later on, it will be joined to the body of the finished expression.

make-assignments (internal)	
vars	a Lisp list of symbols
tmps	a Lisp list of symbols
Returns:	an expression

```
1485 proc ::constcl::make-assignments {vars tmps} {
1486   set res ${::#NIL}
1487   while {$vars ne ${::#NIL}} {
1488     set asg [cons [car $tmps] ${::#NIL}]
1489     set asg [cons [car $vars] $asg]
1490     set asg [cons [S set!] $asg]
1491     set res [cons $asg $res]
1492     set vars [cdr $vars]
1493     set tmps [cdr $tmps]
1494   }
1495   return [cons [S begin] $res]
1496 }
```

make-undefineds procedure

`make-undefineds` creates a list of quoted undefined values.

make-undefineds (internal)	
vals	a Lisp list of values
Returns:	a Lisp list of nil values

```
1497 proc ::constcl::make-undefineds {vals} {
1498   set res ${::#NIL}
1499   while {$vals ne ${::#NIL}} {
1500     set res [cons [list [S quote] ${::#UND}] $res]
1501     set vals [cdr $vals]
1502   }
1503   return $res
1504 }
```

4. Output

The third thing an interpreter must be able to do is to present the resulting code and data so that the user can know what the outcome of the evaluation was.

write procedure

As long as the object given to `write` isn't the empty string, `write` calls the object's `write` method and then writes a newline.

write (public)	
val	a value
?port?	a port
Returns:	nothing

```
1505 reg write
1506
1507 proc ::constcl::write {val args} {
1508   if {$val ne ""} {
1509     set oldport $::constcl::Output_port
1510     if {[llength $args]} {
```

```
1511        lassign $args port
1512        set ::constcl::Output_port $port
1513      }
1514      $val write $::constcl::Output_port
1515      $::constcl::Output_port newline
1516      set ::constcl::Output_port $oldport
1517    }
1518    return
1519  }
```

display procedure

The `display` procedure is like `write` but it calls the object's `display` method and doesn't print a newline afterwards.

display (public)	
val	a value
?port?	a port
Returns:	nothing

```
1520  reg display
1521
1522  proc ::constcl::display {val args} {
1523    if {$val ne ""} {
1524      set oldport $::constcl::Output_port
1525      if {[llength $args]} {
1526        lassign $args port
1527        set ::constcl::Output_port $port
1528      }
1529      $val display $::constcl::Output_port
1530      $::constcl::Output_port flush
1531      set ::constcl::Output_port $oldport
1532    }
```

```
1533    return
1534  }
```

write-pair procedure

The `write-pair` procedure prints a Pair object except for the beginning and ending parentheses.

write-pair (internal)	
port	an output port
pair	a pair
Returns:	nothing

```
1535  proc ::constcl::write-pair {port pair} {
1536    # take an object and print the car
1537    # and the cdr of the stored value
1538    set a [car $pair]
1539    set d [cdr $pair]
1540    # print car
1541    $a write $port
1542    if {[T [pair? $d]]} {
1543      # cdr is a cons pair
1544      $port put " "
1545      write-pair $port $d
1546    } elseif {[T [null? $d]]} {
1547      # cdr is nil
1548      return
1549    } else {
1550      # it is an atom
1551      $port put " . "
1552      $d write $port
1553    }
1554    return
```

```
1555 }
```

5. Identifier validation

idcheckinit procedure
idchecksubs procedure
idcheck procedure
varcheck procedure

Some routines for checking if a string is a valid identifier. `idcheckinit` checks the first character, `idchecksubs` checks the rest. `idcheck` calls the others and raises an error if they fail. A valid symbol is still an invalid identifier if has the name of some keyword, which `varcheck` checks, for a set of keywords given in the standard.

```
1556 proc ::constcl::idcheckinit {init} {
1557   if {[::string is alpha -strict $init] ||
1558     $init in {! $ % & * / : < = > ? ^ _ ~}} {
1559     return true
1560   } else {
1561     return false
1562   }
1563 }
```

```
proc ::constcl::idchecksubs {subs} {
  foreach c [split $subs {}] {
    if {![::string is alnum -strict $c] ||
      $c in {! $ % & * / : < = > ? ^ _ ~ + - . @})} {
      return false
    }
  }
  return true
}
```

```
proc ::constcl::idcheck {sym} {
  if {$sym eq {}} {return $sym}
  if {(![idcheckinit [::string index $sym 0]] ||
    ![idchecksubs [::string range $sym 1 end]]) &&
    $sym ni {+ - ...}} {
    ::error "Identifier expected ($sym)"
  }
  set sym
}
```

```
proc ::constcl::varcheck {sym} {
  if {$sym in {
    else => define unquote unquote-splicing
    quote lambda if set! begin cond and or
    case let let* letrec do delay quasiquote
  }} {
    ::error "Variable name is reserved: $sym"
  }
  return $sym
}
```

6. Environment class and objects

The class for environments is called `Environment`. It is mostly a wrapper around a dictionary, with the added finesse of keeping a link to the outer environment. In this way, there is a chain connecting the latest environment all the way to the global environment and then stopping at the null environment. This chain can be traversed by the `find` method to find which innermost environment a given symbol is bound in.

Using a dictionary means that name lookup is by hash table lookup. In a typical Lisp implementation, large environments are served by hash lookup, while small ones have name lookup by linear search.

The long and complex constructor is to accommodate the variations of Scheme formal parameters lists, which can be an empty list, a proper list, a symbol, or a dotted list.

Names are stored as Lisp symbols, while values are stored as they are, as Lisp or Tcl values. This means that a name might

have to be converted to a symbol before lookup, and the result of lookup may have to be converted afterwards. Note that in the two cases where a number of values are stored under one name (a formals list of a single symbol or a dotted list), then the values are stored as a Lisp list of values.

Environment class

```
1592 oo::class create ::constcl::Environment {
1593   superclass ::constcl::Base
1594   variable bindings outer_env
1595   constructor {syms vals {outer {}}} {
1596     set bindings [dict create]
```

If the formals list (`syms`) is the empty list, then no arguments are accepted.

```
1597     if {[T [::constcl::null? $syms]]} {
1598       if {[llength $vals]} {
1599         error "too many arguments"
1600       }
```

If the formals list is a proper list, there should be one argument per list item.

```
1601     } elseif {[T [::constcl::list? $syms]]} {
1602       set syms [::constcl::splitlist $syms]
1603       set symsn [llength $syms]
1604       set valsn [llength $vals]
1605       if {$symsn != $valsn} {
```

```
1606          error [
1607            ::append --> "wrong # of arguments, " \
1608              "$valsn instead of $symsn"]
1609        }
1610        foreach sym $syms val $vals {
1611          my bind $sym [lindex $val 0] [lindex $val 1]
1612        }
```

If the formals list is actually a single symbol, it takes all the arguments as a list.

```
1613      } elseif {[T [::constcl::symbol? $syms]]} {
1614        my bind $syms VARIABLE [
1615          ::constcl::list {*}[lmap v $vals {
1616            lindex $v 1
1617          }]]
```

Otherwise, bind an argument to the first item in the formals lists and cdr the formals list until the dotted end comes up. Bind all the remaining arguments to it.

```
1618      } else {
1619        while true {
1620          if {[llength $vals] < 1} {
1621            error "too few arguments"
1622          }
1623          my bind [::constcl::car $syms] \
1624            [lindex $vals 0 0] [lindex $vals 0 1]
1625          set vals [lrange $vals 1 end]
1626          if {[T [
1627            ::constcl::symbol? [
1628              ::constcl::cdr $syms]]]} {
1629            my bind [::constcl::cdr $syms] \
```

```
1630              VARIABLE [
1631                ::constcl::list {*}[lmap v $vals {
1632                  lindex $v 1
1633                }]]
1634            set vals {}
1635            break
1636          } else {
1637            set syms [::constcl::cdr $syms]
1638          }
1639        }
1640      }
```

Set the link to the outer environment.

```
1641      set outer_env $outer
1642    }
```

The `find` method searches the environment chain for bindings for a given symbol. The search starts with the current environment instance and ends at the innermost occurrence of a binding for *sym*. The environment containing the binding is returned.

(Environment instance) find (internal)	
sym	a symbol
Returns:	an environment

```
1643    method find {sym} {
1644      ::constcl::check {::constcl::symbol? $sym} {
1645        "SYMBOL expected\nEnvironment find"
1646      }
1647      if {[dict exists $bindings $sym]} {
1648        self
```

```
1649      } else {
1650        $outer_env find $sym
1651      }
1652    }
```

The get method returns the binding type and handling info for *sym* as a tuple.

(Environment instance) get (internal)	
sym	a symbol
Returns:	binding info

```
1653    method get {sym} {
1654      ::constcl::check {::constcl::symbol? $sym} {
1655        "SYMBOL expected\nEnvironment get"
1656      }
1657      dict get $bindings $sym
1658    }
```

The unbind method unsets a binding in the current environment instance. Fails silently if no such binding exists.

(Environment instance) unbind (internal)	
sym	a symbol
Returns:	nothing

```
1659    method unbind {sym} {
1660      ::constcl::check {::constcl::symbol? $sym} {
1661        "SYMBOL expected\nEnvironment unbind"
1662      }
1663      dict unset bindings $sym
1664      return
1665    }
```

The `bind` method binds a symbol in the current environment instance. It is an error to attempt to bind a symbol that is already bound in the environment.

(Environment instance) bind (internal)	
sym	a symbol
type	binding type
info	handling info
Returns:	nothing

```
1666  method bind {sym type info} {
1667    ::constcl::check {::constcl::symbol? $sym} {
1668      "SYMBOL expected\nEnvironment bind"
1669    }
1670    if {[dict exists $bindings $sym]} {
1671      set bi [my get $sym]
1672      lassign $bi bt in
1673      if {$bt in {SPECIAL VARIABLE SYNTAX}} {
1674        error "[$sym name] is already bound"
1675      }
1676    }
1677    dict set bindings $sym [::list $type $info]
1678    return
1679  }
```

The `assign` method updates the location that *sym* is bound to with a new binding type and handling info / value. *Sym* must be bound, and the old binding type must be VARIABLE.

(Environment instance) assign (internal)	
sym	a symbol
type	binding type
info	handling info
Returns:	nothing

```
1680    method assign {sym type info} {
1681      ::constcl::check {::constcl::symbol? $sym} {
1682        "SYMBOL expected\nEnvironment assign"
1683      }
1684      if {![dict exists $bindings $sym]} {
1685        error "[$sym name] is not bound"
1686      }
1687      set bi [my get $sym]
1688      lassign $bi bt in
1689      if {$bt ne "VARIABLE"} {
1690        error "[$sym name] is not assignable"
1691      }
1692      dict set bindings $sym [::list $type $info]
1693      return
1694    }
```

The `parent` method yields the current environment instance's linked outer environment.

(Environment instance) parent (internal)	
Returns:	an environment

```
1695    method parent {} {
1696      set outer_env
1697    }
```

The `names` method returns a Tcl list of all the symbols bound in the current environment instance.

(Environment instance) names (internal)	
Returns:	a Tcl list of symbols

```
1698    method names {} {
1699      dict keys $bindings
1700    }
```

The `values` method returns a Tcl list of all the binding type-/handling info tuples in the current environment instance.

(Environment instance) values (internal)	
Returns:	a Tcl list of binding info tuples

```
1701   method values {} {
1702     dict values $bindings
1703   }
```

The `tstr` method returns an external representation of the environment instance as a Tcl string.

(Environment instance) tstr (internal)	
Returns:	a Tcl string

```
1704   method tstr {} {
1705     regexp {(\d+)} [self] -> num
1706     return "#<env-$num>"
1707   }
1708 }
```

MkEnv generator

The `MkEnv` environment generator can be called with a single argument (the linked-to environment). In that case the parameter and argument lists for the constructor will be empty. If `MkEnv` is called with three arguments, they are, in order, parameters, arguments, and environment.

MkEnv (internal)	
?parms?	a Scheme formals list
?vals?	a Tcl list of values
env	an environment
Returns:	an environment

```
proc ::constcl::MkEnv {args} {
  if {[llength $args] == 1} {
    set parms ${::#NIL}
    set vals {}
    lassign $args env
  } elseif {[llength $args] == 3} {
    lassign $args parms vals env
  } else {
    error "wrong number of arguments"
  }
  Environment new $parms $vals $env
}
```

environment? procedure

Recognizes an environment by type.

environment? (public)	
val	a value
Returns:	a boolean

```
reg environment?

proc ::constcl::environment? {val} {
  typeof? $val Environment
}
```

6.1 Lexical scoping

Example:

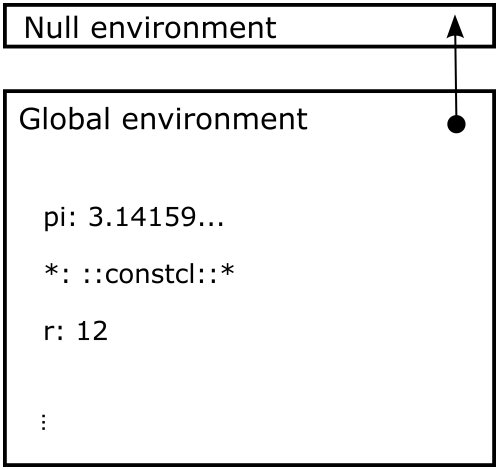
```
ConsTcl> (define (circle-area r) (* pi (* r r)))
ConsTcl> (circle-area 10)
314.1592653589793
```

During a call to the procedure `circle-area`, the symbol `r` is bound to the value 10. But we don't want the binding to go into the global environment, possibly clobbering an earlier definition of `r`. The solution is to use separate (but linked) environments, making `r`'s binding a local variable[14] in its own environment, which the procedure will be evaluated in. The symbols `*` and `pi` will still be available through the local environment's link to the outer global environment. This is all part of lexical scoping[15].

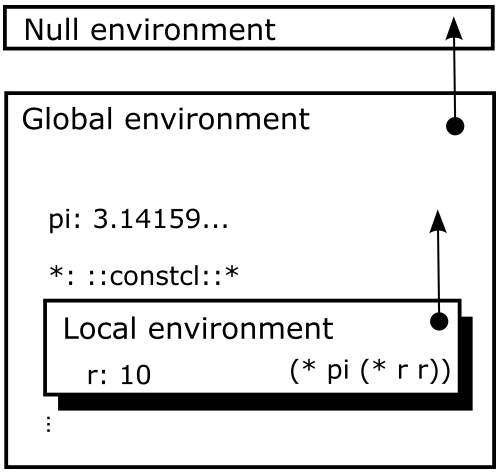
In the first image, we see the global environment before we call `circle-area` (and also the empty null environment which the global environment links to):

[14] See `https://en.wikipedia.org/wiki/Local_variable`

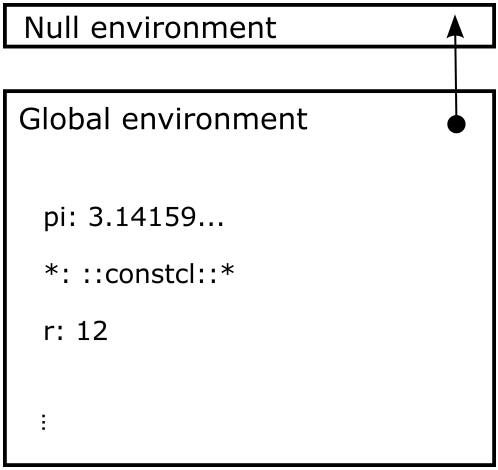
[15] See `https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scope`



During the call. Note how the global `r` is shadowed by the local one, and how the local environment links to the global one to find `*` and `pi`.



After the call, we are back to the first state again.


Null environment
Global environment
pi: 3.14159...
: ::constcl::
r: 12

7. The REPL

The REPL (read-eval-print loop) is a loop that repeatedly *reads* a Scheme source string from the user through the command `::constcl::input` (breaking the loop if given an empty line) and `::constcl::parse`, *evaluates* it using `::constcl::eval`, and *prints* using `::constcl::write`.

input

`input` is modelled after the Python 3 function. It displays a prompt and reads a string.

```
1726 proc ::constcl::input {prompt} {
1727   puts -nonewline $prompt
1728   flush stdout
1729   set buf [gets stdin]
1730   set openpars [regexp -all -inline {\(} $buf]
1731   set clsepars [regexp -all -inline {\)} $buf]
1732   set openbrak [regexp -all -inline {\[} $buf]
1733   set clsebrak [regexp -all -inline {\]} $buf]
1734   while {[llength $openpars] > [llength $clsepars] ||
1735          [llength $openbrak] > [llength $clsebrak]} {
1736     ::append buf [gets stdin]
1737     set openpars [regexp -all -inline {\(} $buf]
```

```
1738      set clsepars [regexp -all -inline {\)} $buf]
1739      set openbrak [regexp -all -inline {\[} $buf]
1740      set clsebrak [regexp -all -inline {\]} $buf]
1741    }
1742    return $buf
1743  }
```

repl

`repl` puts the 'loop' in the read-eval-print loop. It repeats prompting for a string until given a blank input. Given non-blank input, it parses and evaluates the string, printing the resulting value.

```
1744  proc ::repl {{prompt "ConsTcl> "}} {
1745    set cur_env [::constcl::MkEnv ::constcl::global_env]
1746    set str [::constcl::input $prompt]
1747    while {$str ne ""} {
1748      set expr [parse $str]
1749      set val [::constcl::eval $expr $cur_env]
1750      ::constcl::write $val
1751      set str [::constcl::input $prompt]
1752    }
1753    $cur_env destroy
1754  }
```

Well!

After 1754 lines of code, the interpreter is done.

Now for the built-in types and procedures!

Part II

Built-in types and procedures

8. The standard library

8.1 Equivalence predicates

One of the fundamental questions in programming is "is A equal to B?". Lisp takes the question and adds "what does it mean to be equal?"

Lisp has a number of equivalence predicates. ConsTcl, like Scheme, has three. Of the three, `eq?` generally tests for identity (with exception for numbers), `eqv?` tests for value equality (except for booleans and procedures, where it tests for identity), and `equal?` tests for whether the output strings are equal.

eq? procedure

eq?, eqv?, equal? (public)	
expr1	an expression
expr2	an expression
Returns:	a boolean

```
1755 reg eq?
1756
1757 proc ::constcl::eq? {expr1 expr2} {
1758   if {[teq boolean? $expr1 $expr2] &&
1759       $expr1 eq $expr2} {
1760     return ${::#t}
1761   } elseif {[teq symbol? $expr1 $expr2] &&
1762       $expr1 eq $expr2} {
1763     return ${::#t}
1764   } elseif {[teq number? $expr1 $expr2] &&
1765       [veq $expr1 $expr2]} {
1766     return ${::#t}
1767   } elseif {[teq char? $expr1 $expr2] &&
1768       $expr1 eq $expr2} {
1769     return ${::#t}
1770   } elseif {[teq null? $expr1 $expr2]} {
1771     return ${::#t}
1772   } elseif {[teq pair? $expr1 $expr2] &&
1773       $expr1 eq $expr2} {
1774     return ${::#t}
1775   } elseif {[teq string? $expr1 $expr2] &&
1776       $expr1 eq $expr2} {
1777     return ${::#t}
1778   } elseif {[teq vector? $expr1 $expr2] &&
1779       $expr1 eq $expr2} {
1780     return ${::#t}
1781   } elseif {[teq procedure? $expr1 $expr2] &&
1782       $expr1 eq $expr2} {
1783     return ${::#t}
1784   } else {
1785     return ${::#f}
1786   }
1787 }
```

---

teq procedure

`teq` tests for type equality, i.e. that the expressions have the same type.

teq (internal)	
typep	a procedure
expr1	an expression
expr2	an expression
Returns:	a Tcl truth value (1 or 0)

```
1788 proc ::constcl::teq {typep expr1 expr2} {
1789     return [expr {[T [$typep $expr1]] &&
1790       [T [$typep $expr2]]}]
1791 }
```

veq procedure

`veq` tests for value equality, i.e. that the expressions have the same value.

veq (internal)	
expr1	an expression
expr2	an expression
Returns:	a Tcl truth value (1 or 0)

```
1792 proc ::constcl::veq {expr1 expr2} {
1793     return [expr {[$expr1 value] eq [$expr2 value]}]
1794 }
```

eqv? procedure

```
1795 reg eqv?
```

```
1796
1797 proc ::constcl::eqv? {expr1 expr2} {
1798   if {[teq boolean? $expr1 $expr2] &&
1799       $expr1 eq $expr2} {
1800     return ${::#t}
1801   } elseif {[teq symbol? $expr1 $expr2] &&
1802       [veq $expr1 $expr2]} {
1803     return ${::#t}
1804   } elseif {[teq number? $expr1 $expr2] &&
1805       [veq $expr1 $expr2]} {
1806     return ${::#t}
1807   } elseif {[teq char? $expr1 $expr2] &&
1808       [veq $expr1 eq $expr2]} {
1809     return ${::#t}
1810   } elseif {[teq null? $expr1 $expr2]} {
1811     return ${::#t}
1812   } elseif {[T [pair? $expr1]] &&
1813       [T [pair? $expr2]] &&
1814       [$expr1 car] eq [$expr2 car] &&
1815       [$expr1 cdr] eq [$expr2 cdr]} {
1816     return ${::#t}
1817   } elseif {[teq string? $expr1 $expr2] &&
1818       [veq $expr1 $expr2]} {
1819     return ${::#t}
1820   } elseif {[teq vector? $expr1 $expr2] &&
1821       [veq $expr1 $expr2]} {
1822     return ${::#t}
1823   } elseif {[teq procedure? $expr1 $expr2] &&
1824       $expr1 eq $expr2} {
1825     return ${::#t}
1826   } else {
1827     return ${::#f}
1828   }
1829 }
```

equal? procedure

```
reg equal?

proc ::constcl::equal? {expr1 expr2} {
  if {[$expr1 tstr] eq [$expr2 tstr]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

8.2 Numbers

The word 'computer' suggests numerical calculations. A programming language is almost no use if it doesn't support at least arithmetic. Scheme has a rich numerical library and many number types that support advanced calculations.

I have only implemented a bare-bones version of Scheme's numerical library, though. The following is a reasonably complete framework for operations on integers and floating-point numbers. No rationals, no complex numbers, no `gcd` or `lcm`.

Number class

The Number class defines what capabilities a number has (in addition to those from the Base class), and also defines the internal representation of a number value expression. A number

is stored in an instance in Tcl form, and the `numval` method yields the Tcl number as result.

```
1839 oo::class create ::constcl::Number {
1840   superclass ::constcl::Base
1841   variable value
```

The constructor tests its argument against the form of a double-precision floating point number, which admits an integer number as well.

Number constructor (internal)	
val	an external representation of a number
Returns:	nothing

```
1842   constructor {val} {
1843     if {[::string is double -strict $val]} {
1844       set value $val
1845     } else {
1846       ::error "NUMBER expected\n$val"
1847     }
1848   }
```

The `zero?` method is a predicate that tells if the stored number is equal to 0.

(Number instance) zero? (internal)	
Returns:	a boolean

```
1849   method zero? {} {
1850     if {$value == 0} {
1851       return ${::#t}
1852     } else {
```

```
1853        return ${::#f}
1854      }
1855    }
```

The `positive?` method is a predicate that tells if the stored number is greater than 0.

(Number instance) positive? (internal)	
Returns:	a boolean

```
1856    method positive? {} {
1857      if {$value > 0} {
1858        return ${::#t}
1859      } else {
1860        return ${::#f}
1861      }
1862    }
```

The `negative?` method is a predicate that tells if the stored number is less than 0.

(Number instance) negative? (internal)	
Returns:	a boolean

```
1863    method negative? {} {
1864      if {$value < 0} {
1865        return ${::#t}
1866      } else {
1867        return ${::#f}
1868      }
1869    }
```

The `even?` method is a predicate that tells if the stored number is even.

(Number instance) even? (internal)	
Returns:	a boolean

```
1870  method even? {} {
1871    if {$value % 2 == 0} {
1872      return ${::#t}
1873    } else {
1874      return ${::#f}
1875    }
1876  }
```

The `odd?` method is a predicate that tells if the stored number is odd.

(Number instance) odd? (internal)	
Returns:	a boolean

```
1877  method odd? {} {
1878    if {$value % 2 == 1} {
1879      return ${::#t}
1880    } else {
1881      return ${::#f}
1882    }
1883  }
```

The `value` method returns the stored number.

(Number instance) value (internal)	
Returns:	a number

```
1884  method value {} {
1885    set value
1886  }
```

The `numval` method is a synonym for `value`.

(Number instance) numval (internal)	
Returns:	a number

```
1887    method numval {} {
1888      set value
1889    }
```

The `constant` method signals that the number instance isn't mutable.

(Number instance) constant (internal)	
Returns:	a Tcl truth value (1)

```
1890    method constant {} {
1891      return 1
1892    }
```

The `tstr` method yields the external representation of the stored value as a Tcl string. It is used by error messages and the `write` method.

(Char instance) tstr (internal)	
Returns:	an external representation of a number

```
1893    method tstr {} {
1894      return $value
1895    }
1896  }
```

MkNumber generator

`MkNumber` generates a Number object. Short form: `N`.

MkNumber (internal)	
str	a string
Returns:	a number

```
interp alias {} ::constcl::MkNumber \
  {} ::constcl::Number new
interp alias {} N {} ::constcl::Number new
```

number? procedure

`number?` recognizes a number by object type, not by content.

number? (public)	
val	a value
Returns:	a boolean

```
reg number?

proc ::constcl::number? {val} {
  return [typeof? $val Number]
}
```

= procedure

< procedure
> procedure
<= procedure

>= procedure

The predicates =, <, >, <=, and >= are implemented.

=, <, >, <=, >= (public)	
nums	some numbers
Returns:	a boolean

```
reg =

proc ::constcl::= {args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(= [
      [lindex $args 0] tstr] ...)"
  }
  if {[::tcl::mathop::== {*}$nums]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg <

proc ::constcl::< {args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(< num ...)"
  }
  if {[::tcl::mathop::< {*}$nums]} {
    return ${::#t}
  } else {
```

```
    return ${::#f}
  }
}
```

```
reg >

proc ::constcl::> {args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(> num ...)"
  }
  if {[::tcl::mathop::> {*}$nums]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg <=

proc ::constcl::<= {args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(<= num ...)"
  }
  if {[::tcl::mathop::<= {*}$nums]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg >=

proc ::constcl::>= {args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(>= num ...)"
  }
  if {[::tcl::mathop::>= {*}$nums]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

zero? procedure

The zero? predicate tests if a given number is equal to zero.

zero? (public)	
num	a number
Returns:	a boolean

```
reg zero?

proc ::constcl::zero? {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  return [$num zero?]
}
```

positive? procedure

negative? procedure
even? procedure
odd? procedure

The `positive?/negative?/even?/odd?` predicates test a number for those traits.

positive?, negative?, even?, odd? (public)	
num	a number
Returns:	a boolean

```
1984 reg positive?
1985
1986 proc ::constcl::positive? {num} {
1987   check {number? $num} {
1988       NUMBER expected\n([pn] [$num tstr])
1989   }
1990   return [$num positive?]
1991 }
```

```
1992 reg negative?
1993
1994 proc ::constcl::negative? {num} {
1995   check {number? $num} {
1996       NUMBER expected\n([pn] [$num tstr])
1997   }
1998   return [$num negative?]
1999 }
```

```
2000 reg even?
```

```
2001
2002 proc ::constcl::even? {num} {
2003   check {number? $num} {
2004       NUMBER expected\n([pn] [$num tstr])
2005   }
2006   return [$num even?]
2007 }
```

```
2008 reg odd?
2009
2010 proc ::constcl::odd? {num} {
2011   check {number? $num} {
2012       NUMBER expected\n([pn] [$num tstr])
2013   }
2014   return [$num odd?]
2015 }
```

max procedure

min procedure

The `max` function selects the largest number, and the `min` function selects the smallest number.

Example:

```
(max 7 1 10 3)   =>  10
(min 7 1 10 3)   =>  1
```

max, min (public)	
num	a number
nums	some numbers
Returns:	a number

```
reg max

proc ::constcl::max {num args} {
  lappend args $num
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(max num...)"
  }
  N [::tcl::mathfunc::max {*}$nums]
}
```

```
reg min

proc ::constcl::min {num args} {
  lappend args $num
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(min num...)"
  }
  N [::tcl::mathfunc::min {*}$nums]
}
```

+ procedure

***** procedure
- procedure
/ procedure

The operators +, *, -, and / stand for the respective arithmetic operations. They take a number of operands, but at least one for - and /.

Example:

```
(list (+ 2 2) (* 2 2) (- 10 6) (/ 20 5))  => (4 4 4 4)
(+ 21 7 3)                                 => 31
(* 21 7 3)                                 => 441
(- 21 7 3)                                 => 11
(/ 21 7 3)                                 => 1
(- 5)                                      => -5
(/ 5)                                      => 0.2
```

+, * (public)	
?nums?	some numbers
Returns:	a number

-, / (public)	
num	a number
?nums?	some numbers
Returns:	a number

```
reg +

proc ::constcl::+ {args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(+ num ...)"
  }
  N [::tcl::mathop::+ {*}$nums]
}
```

```
reg *

proc ::constcl::* {args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(* num ...)"
  }
  N [::tcl::mathop::* {*}$nums]
}
```

```
reg -

proc ::constcl::- {num args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(- num ...)"
  }
  N [::tcl::mathop::- [$num numval] {*}$nums]
}
```

```
reg /

proc ::constcl::/ {num args} {
  try {
    set nums [lmap arg $args {$arg numval}]
  } on error {} {
    ::error "NUMBER expected\n(/ num ...)"
  }
  N [::tcl::mathop::/ [$num numval] {*}$nums]
}
```

abs procedure

The abs function yields the absolute value of a number.

abs (public)	
num	a number
Returns:	a number

```
2078 reg abs
2079
2080 proc ::constcl::abs {num} {
2081   check {number? $num} {
2082       NUMBER expected\n([pn] [$num tstr])
2083   }
2084   if {[T [$num negative?]]} {
2085     return [N [expr {[$num numval] * -1}]]
2086   } else {
2087     return $num
2088   }
2089 }
```

quotient procedure

quotient calculates the quotient between two numbers.
Example:

```
(quotient 7 3)   =>  2.0
```

quotient (public)	
num1	a number
num2	a number
Returns:	a number

```
reg quotient

proc ::constcl::quotient {num1 num2} {
  set q [::tcl::mathop::/ [$num1 numval] \
    [$num2 numval]]
  if {$q > 0} {
    return [N [::tcl::mathfunc::floor $q]]
  } elseif {$q < 0} {
    return [N [::tcl::mathfunc::ceil $q]]
  } else {
    return [N 0]
  }
}
```

remainder procedure

`remainder` is similar to `modulo`, but the remainder is calculated using absolute values for `num1` and `num2`, and the result is negative if and only if `num1` was negative.

Example:

```
(remainder 7 3)   =>  1
```

remainder (public)	
num1	a number
num2	a number
Returns:	a number

```
reg remainder

proc ::constcl::remainder {num1 num2} {
```

```
2106    set n [::tcl::mathop::% [[abs $num1] numval] \
2107      [[abs $num2] numval]]
2108    if {[T [$num1 negative?]]} {
2109      set n -$n
2110    }
2111    return [N $n]
2112  }
```

modulo procedure

Example:

```
(modulo 7 3)   =>  1
```

modulo (public)	
num1	a number
num2	a number
Returns:	a number

```
2113  reg modulo
2114
2115  proc ::constcl::modulo {num1 num2} {
2116    return [N [::tcl::mathop::% [$num1 numval] \
2117      [$num2 numval]]]
2118  }
```

floor procedure

ceiling procedure
truncate procedure

round procedure

`floor`, `ceiling`, `truncate`, and `round` are different methods for converting a floating point number to an integer.

Example:

```
(floor 7.5)      =>  7.0
(ceiling 7.5)    =>  8.0
(truncate 7.5)   =>  7.0
(round 7.5)      =>  8
```

floor, ceiling, truncate, round (public)	
num	a number
Returns:	a number

```
reg floor

proc ::constcl::floor {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::floor [$num numval]]
}
```

```
reg ceiling

proc ::constcl::ceiling {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::ceil [$num numval]]
}
```

```
reg truncate

proc ::constcl::truncate {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  if {[T [$num negative?]]} {
    N [::tcl::mathfunc::ceil [$num numval]]
  } else {
    N [::tcl::mathfunc::floor [$num numval]]
  }
}
```

```
reg round

proc ::constcl::round {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::round [$num numval]]
}
```

exp procedure

log procedure
sin procedure
cos procedure
tan procedure
asin procedure
acos procedure

atan procedure

The mathematical functions e^x, natural logarithm, sine, cosine, tangent, arcsine, arccosine, and arctangent are calculated by `exp`, `log`, `sin`, `cos`, `tan`, `asin`, `acos`, and `atan`, respectively. `atan` can be called both as a unary (one argument) function and a binary (two arguments) one.

Example:

```
(let ((x (log 2))) (= 2 (exp x)))   =>  #t
(let* ((a (/ pi 3)) (s (sin a)))
  (= a (asin s)))                   =>  #t
```

exp, log, sin, cos, tan, asin, acos, atan (public)	
num	a number
Returns:	a number

(binary) atan (public)	
num1	a number
num2	a number
Returns:	a number

```
reg exp

proc ::constcl::exp {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::exp [$num numval]]
}
```

```
reg log

proc ::constcl::log {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::log [$num numval]]
}
```

```
reg sin

proc ::constcl::sin {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::sin [$num numval]]
}
```

```
reg cos

proc ::constcl::cos {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::cos [$num numval]]
}
```

```
reg tan

proc ::constcl::tan {num} {
```

```
2190    check {number? $num} {
2191        NUMBER expected\n([pn] [$num tstr])
2192    }
2193    N [::tcl::mathfunc::tan [$num numval]]
2194  }
```

```
2195  reg asin
2196
2197  proc ::constcl::asin {num} {
2198    check {number? $num} {
2199        NUMBER expected\n([pn] [$num tstr])
2200    }
2201    N [::tcl::mathfunc::asin [$num numval]]
2202  }
```

```
2203  reg acos
2204
2205  proc ::constcl::acos {num} {
2206    check {number? $num} {
2207        NUMBER expected\n([pn] [$num tstr])
2208    }
2209    N [::tcl::mathfunc::acos [$num numval]]
2210  }
```

```
2211  reg atan
2212
2213  proc ::constcl::atan {args} {
2214    if {[llength $args] == 1} {
2215      set num [lindex $args 0]
2216      check {number? $num} {
```

```
2217          NUMBER expected\n([pn] [$num tstr])
2218      }
2219      N [::tcl::mathfunc::atan [$num numval]]
2220    } else {
2221      lassign $args num1 num2
2222      check {number? $num1} {
2223          NUMBER expected\n([pn] [$num1 tstr])
2224      }
2225      check {number? $num2} {
2226          NUMBER expected\n([pn] [$num2 tstr])
2227      }
2228      N [::tcl::mathfunc::atan2 \
2229        [$num1 numval] [$num2 numval]]
2230    }
2231  }
```

sqrt procedure

sqrt calculates the square root.

sqrt (public)	
num	a number
Returns:	a number

```
2232  reg sqrt
2233
2234  proc ::constcl::sqrt {num} {
2235    check {number? $num} {
2236        NUMBER expected\n([pn] [$num tstr])
2237    }
2238    N [::tcl::mathfunc::sqrt [$num numval]]
2239  }
```

expt procedure

`expt` calculates x to the power y, or x^y.

expt (public)	
x	a number
y	a number
Returns:	a number

```
reg expt

proc ::constcl::expt {x y} {
  check {number? $x} {
      NUMBER expected\n([pn] [$x tstr] \
        [$y tstr])
  }
  check {number? $y} {
      NUMBER expected\n([pn] [$x tstr] \
        [$y tstr])
  }
  N [::tcl::mathfunc::pow [$x numval] \
    [$y numval]]
}
```

number->string procedure

The procedures `number->string` and `string->number` convert between number and string with optional radix conversion.

Example:

```
(number->string 23)      =>  "23"
(number->string 23 2)    =>  "10111"
```

```
(number->string 23 8)    =>  "27"
(number->string 23 16)   =>  "17"
```

number->string (public)	
num	a number
?radix?	a number
Returns:	a string

```
reg number->string

proc ::constcl::number->string {num args} {
  if {[llength $args] == 0} {
    check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
    }
    return [MkString [$num numval]]
  } else {
    lassign $args radix
    check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
    }
    check {number? $radix} {
      NUMBER expected\n([pn] [$num tstr] \
        [$radix tstr])
    }
    set radices [list [N 2] [N 8] [N 10] [N 16]]
    check {memv $radix $radices} {
      Radix not in 2, 8, 10, 16\n([pn] \
        [$num tstr] [$radix tstr])
    }
    if {[$radix numval] == 10} {
      return [MkString [$num numval]]
    } else {
      return [MkString [base [$radix numval] \
        [$num numval]]]
```

```
    }
  }
}
```

base is due to Richard Suchenwirth[16].

```
proc base {base number} {
  set negative [regexp ^-(.+) $number -> number]
  set digits {0 1 2 3 4 5 6 7 8 9 A B C D E F}
  set res {}
  while {$number} {
    set digit [expr {$number % $base}]
    set res [lindex $digits $digit]$res
    set number [expr {$number / $base}]
  }
  if $negative {set res -$res}
  set res
}
```

string->number procedure

As with number->string, above.

Example:

```
(string->number "23")        =>  23
(string->number "10111" 2)   =>  23
(string->number "27" 8)      =>  23
(string->number "17" 16)     =>  23
```

[16] See https://wiki.tcl-lang.org/page/Based+numbers

string->number (public)	
str	a string
?radix?	a number
Returns:	a number

```
reg string->number

proc ::constcl::string->number {str args} {
  if {[llength $args] == 0} {
    check {string? $str} {
      STRING expected\n([pn] [$str tstr])
    }
    return [N [$str value]]
  } else {
    lassign $args radix
    check {string? $str} {
      STRING expected\n([pn] [$str tstr])
    }
    set radices [list [N 2] [N 8] [N 10] [N 16]]
    check {memv $radix $radices} {
      Radix not in 2, 8, 10, 16\n([pn] [$str tstr] \
        [$radix tstr])
    }
    if {[$radix numval] == 10} {
      return [N [$str value]]
    } else {
      return [N [
        frombase [$radix numval] [$str value]]]
    }
  }
}
```

`frombase` is due to Richard Suchenwirth[17].

[17] See `https://wiki.tcl-lang.org/page/Based+numbers`

```
2322 proc frombase {base number} {
2323   set digits {0 1 2 3 4 5 6 7 8 9 A B C D E F}
2324   set negative [regexp ^-(.+) $number -> number]
2325   set res 0
2326   foreach digit [split $number {}] {
2327     # dv = decimal value
2328     set dv [lsearch $digits $digit]
2329     if {$dv < 0 || $dv >= $base} {
2330       ::error "bad digit $dv for base $base"
2331     }
2332     set res [expr {$res * $base + $dv}]
2333   }
2334   if $negative {set res -$res}
2335   set res
2336 }
```

8.3 Booleans

Booleans are logic values, either true (`#t`) or false (`#f`). All predicates (procedures whose name end with -?) return boolean values.

Pseudo-booleans

All values can be tested for truth (in a conditional form or as arguments to `and`, `or`, or `not`), though. Any value of any type is considered to be true except for `#f`.

Boolean classes (True and False)

The Boolean classes are singleton classes with one value each (see page 303) (the global values `#t` and `#f`, respectively).

```
2337 oo::singleton create ::constcl::True {
2338   superclass ::constcl::Base
```

The `tstr` method yields the value `#t` as a Tcl string. It is used for error messages.

(True instance) tstr (internal)	
Returns:	the external representation of true

```
2339   method tstr {} {
2340     return "#t"
2341   }
2342 }
```

```
2343 oo::singleton create ::constcl::False {
2344   superclass ::constcl::Base
```

The `tstr` method yields the value `#f` as a Tcl string. It is used for error messages.

(False instance) tstr (internal)	
Returns:	the external representation of false

```
2345   method tstr {} {
2346     return "#f"
2347   }
2348 }
```

MkBoolean generator

Given a string (either `"#t"` or `"#f"`), `MkBoolean` generates a boolean.

MkBoolean (internal)	
bool	an external representation of a bool
Returns:	a boolean

```
2349 proc ::constcl::MkBoolean {bool} {
2350   switch $bool {
2351     "#t" { return ${::#t} }
2352     "#f" { return ${::#f} }
2353     default { ::error "invalid boolean ($bool)" }
2354   }
2355 }
```

boolean? procedure

The `boolean?` predicate recognizes a boolean by object identity (i.e. is it the true or false constant? If yes, then it is a boolean).

boolean? (public)	
val	a value
Returns:	a boolean

```
2356 reg boolean?
2357
2358 proc ::constcl::boolean? {val} {
2359   if {$val eq ${::#t} || $val eq ${::#f}} {
2360     return ${::#t}
2361   } else {
2362     return ${::#f}
```

```
2363    }
2364  }
```

not procedure

The only operations on booleans are the macros `and` and `or` (see page 99), and `not` (logical negation).

Example:

```
(not #f)    ==>  #t   ; #f yields #t, all others #f
(not nil)   ==>  #f   ; see?
```

not (public)	
val	a value
Returns:	a boolean

```
2365  reg not
2366
2367  proc ::constcl::not {val} {
2368    if {$val eq ${::#f}} {
2369      return ${::#t}
2370    } else {
2371      return ${::#f}
2372    }
2373  }
```

8.4 Characters

Characters are any Unicode graphic character, and also space and newline space characters. External representation is `#\A` (A stands for any character) or `#\space` or `#\newline`.

Char class

The Char class defines what capabilities a character has (in addition to those from the Base class), and also defines the internal representation of a character value expression. A character is stored in an instance as a Tcl character, and the `char` method yields the character as result.

```
2374 oo::class create ::constcl::Char {
2375   superclass ::constcl::Base
2376   variable value
```

The constructor tests its argument against the three basic forms of external representation for characters, and stores the corresponding Tcl character.

Char constructor (internal)	
val	an external representation of a char
Returns:	nothing

```
2377   constructor {val} {
2378     switch -regexp $val {
2379       {(?i)#\\space} {
2380         set val " "
2381       }
2382       {(?i)#\\newline} {
```

```
2383          set val "\n"
2384        }
2385        {#\\[[:graph:]]} {
2386          set val [::string index $val 2]
2387        }
2388        default {
2389          ::error "CHAR expected\n$val"
2390        }
2391      }
2392      set value $val
2393    }
```

The `char` method yields the stored character value.

(Char instance) char (internal)	
Returns:	a Tcl character

```
2394    method char {} {
2395      set value
2396    }
```

The `alphabetic?` method is a predicate which tests if the stored value is an alphabetic character.

(Char instance) alphabetic? (internal)	
Returns:	a boolean

```
2397    method alphabetic? {} {
2398      if {[::string is alpha $value]} {
2399        return ${::#t}
2400      } else {
2401        return ${::#f}
2402      }
2403    }
```

The `numeric?` method is a predicate which tests if the stored value is a numeric character.

(Char instance) numeric? (internal)	
Returns:	a boolean

```
2404  method numeric? {} {
2405    if {[::string is digit $value]} {
2406      return ${::#t}
2407    } else {
2408      return ${::#f}
2409    }
2410  }
```

The `whitespace?` method is a predicate which tests if the stored value is a whitespace character.

(Char instance) whitespace? (internal)	
Returns:	a boolean

```
2411  method whitespace? {} {
2412    if {[::string is space $value]} {
2413      return ${::#t}
2414    } else {
2415      return ${::#f}
2416    }
2417  }
```

The `upper-case?` method is a predicate which tests if the stored value is an uppercase character.

(Char instance) upper-case? (internal)	
Returns:	a boolean

```
2418   method upper-case? {} {
2419     if {[::string is upper $value]} {
2420       return ${::#t}
2421     } else {
2422       return ${::#f}
2423     }
2424   }
```

The `lower-case?` method is a predicate which tests if the stored value is an lowercase character.

(Char instance) lower-case? (internal)	
Returns:	a boolean

```
2425   method lower-case? {} {
2426     if {[::string is lower $value]} {
2427       return ${::#t}
2428     } else {
2429       return ${::#f}
2430     }
2431   }
```

The `constant` method signals that the character instance isn't mutable.

(Char instance) constant (internal)	
Returns:	a Tcl truth value (1)

```
2432   method constant {} {
2433     return 1
2434   }
```

The `value` method is another way to yield the stored value

(Char instance) value (internal)	
Returns:	a Tcl character

```
2435   method value {} {
2436     return $value
2437   }
```

The `external` method translates the stored value back to external representation.

(Char instance) external (internal)	
Returns:	an external representation of a char

```
2438   method external {} {
2439     switch $value {
2440       " " {
2441         return "#\\space"
2442       }
2443       "\n" {
2444         return "#\\newline"
2445       }
2446       default {
2447         return "#\\$value"
2448       }
2449     }
2450   }
```

The `display` method is used by the `display` standard procedure to print the stored value as a character.

(Char instance) display (internal)	
port	an output port
Returns:	nothing

```
2451   method display {port} {
2452     $port put [my char]
2453   }
```

The `tstr` method yields the external representation of the stored value as a Tcl string. It is used by error messages and the `write` method.

(Char instance) tstr (internal)	
Returns:	an external representation of a char

```
2454   method tstr {} {
2455     return [my external]
2456   }
2457 }
```

MkChar generator

`MkChar` generates a character object. If a character object with the same name already exists, that character will be returned, otherwise a fresh character will be created.

MkChar (internal)	
char	an external representation of a char
Returns:	a character

```
2458 proc ::constcl::MkChar {char} {
2459   if {[regexp -nocase {space|newline} $char]} {
2460       set char [::string tolower $char]
2461   }
2462   foreach instance [
2463     info class instances Char] {
2464     if {[$instance external] eq $char} {
2465       return $instance
2466     }
2467   }
2468   return [::constcl::Char new $char]
2469 }
```

char? procedure

`char?` recognizes Char values by type.

char? (public)	
val	a value
Returns:	a boolean

```
2470 reg char?
2471
2472 proc ::constcl::char? {val} {
2473   return [typeof? $val Char]
2474 }
```

char=? procedure

char<? procedure
char>? procedure
char<=? procedure
char>=? procedure

`char=?`, `char<?`, `char>?`, `char<=?`, and `char>=?` compare character values. They only compare two characters at a time.

char=?, char<?, char>? (public)	
char1	a character
char2	a character
Returns:	a boolean

char<=?, char>=? (public)	
char1	a character
char2	a character
Returns:	a boolean

```
reg char=?

proc ::constcl::char=? {char1 char2} {
  check {char? $char1} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  check {char? $char2} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  if {[$char1 char] eq [$char2 char]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg char<?

proc ::constcl::char<? {char1 char2} {
  check {char? $char1} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  check {char? $char2} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  if {[$char1 char] < [$char2 char]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg char>?

proc ::constcl::char>? {char1 char2} {
  check {char? $char1} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  check {char? $char2} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  if {[$char1 char] > [$char2 char]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg char<=?

proc ::constcl::char<=? {char1 char2} {
  check {char? $char1} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  check {char? $char2} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  if {[$char1 char] <= [$char2 char]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg char>=?

proc ::constcl::char>=? {char1 char2} {
  check {char? $char1} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  check {char? $char2} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  if {[$char1 char] >= [$char2 char]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

char-ci=? procedure

char-ci<? procedure
char-ci>? procedure
char-ci<=? procedure
char-ci>=? procedure

`char-ci=?`, `char-ci<?`, `char-ci>?`, `char-ci<=?`, and `char-ci>=?` compare character values in a case insensitive manner. They only compare two characters at a time.

char-ci=?, char-ci<?, char-ci>? (public)	
char1	a character
char2	a character
Returns:	a boolean

char-ci<=?, char-ci>=? (public)	
char1	a character
char2	a character
Returns:	a boolean

```
reg char-ci=?

proc ::constcl::char-ci=? {char1 char2} {
  check {char? $char1} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  check {char? $char2} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  if {[::string tolower [$char1 char]] eq
      [::string tolower [$char2 char]]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg char-ci<?

proc ::constcl::char-ci<? {char1 char2} {
  check {char? $char1} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  check {char? $char2} {
    CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
  }
  if {[::string tolower [$char1 char]] <
      [::string tolower [$char2 char]]} {
```

```
2577      return ${::#t}
2578    } else {
2579      return ${::#f}
2580    }
2581  }
```

```
2582  reg char-ci>?
2583
2584  proc ::constcl::char-ci>? {char1 char2} {
2585    check {char? $char1} {
2586      CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
2587    }
2588    check {char? $char2} {
2589      CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
2590    }
2591    if {[::string tolower [$char1 char]] >
2592        [::string tolower [$char2 char]]} {
2593      return ${::#t}
2594    } else {
2595      return ${::#f}
2596    }
2597  }
```

```
2598  reg char-ci<=?
2599
2600  proc ::constcl::char-ci<=? {char1 char2} {
2601    check {char? $char1} {
2602      CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
2603    }
2604    check {char? $char2} {
2605      CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
2606    }
```

```
2607    if {[::string tolower [$char1 char]] <=
2608        [::string tolower [$char2 char]]} {
2609      return ${::#t}
2610    } else {
2611      return ${::#f}
2612    }
2613  }
```

```
2614  reg char-ci>=?
2615
2616  proc ::constcl::char-ci>=? {char1 char2} {
2617    check {char? $char1} {
2618      CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
2619    }
2620    check {char? $char2} {
2621      CHAR expected\n([pn] [$char1 tstr] [$char2 tstr])
2622    }
2623    if {[::string tolower [$char1 char]] >=
2624        [::string tolower [$char2 char]]} {
2625      return ${::#t}
2626    } else {
2627      return ${::#f}
2628    }
2629  }
```

char-alphabetic? procedure

char-numeric? procedure
char-whitespace? procedure
char-upper-case? procedure
char-lower-case? procedure

The predicate `char-alphabetic?` tests if a character is alphabetic, `char-numeric?` tests if a character is numeric, and `char-whitespace?` tests if a character is whitespace. `char-upper-case?` and `char-lower-case?` test if a character is upper- or lower-case.

char-alphabetic?, char-numeric? (public)	
char	a character
Returns:	a boolean

char-whitespace? (public)	
char	a character
Returns:	a boolean

char-upper-case?, char-lower-case? (public)	
char	a character
Returns:	a boolean

```
2630 reg char-alphabetic?
2631
2632 proc ::constcl::char-alphabetic? {char} {
2633   check {char? $char} {
2634     CHAR expected\n([pn] [$char tstr])
2635   }
2636   return [$char alphabetic?]
2637 }
```

```
2638 reg char-numeric?
2639
2640 proc ::constcl::char-numeric? {char} {
2641   check {char? $char} {
2642     CHAR expected\n([pn] [$char tstr])
2643   }
```

```
2644    return [$char numeric?]
2645  }
```

```
2646  reg char-whitespace?
2647
2648  proc ::constcl::char-whitespace? {char} {
2649    check {char? $char} {
2650      CHAR expected\n([pn] [$char tstr])
2651    }
2652    return [$char whitespace?]
2653  }
```

```
2654  reg char-upper-case?
2655
2656  proc ::constcl::char-upper-case? {char} {
2657    check {char? $char} {
2658      CHAR expected\n([pn] [$char tstr])
2659    }
2660    return [$char upper-case?]
2661  }
```

```
2662  reg char-lower-case?
2663
2664  proc ::constcl::char-lower-case? {char} {
2665    check {char? $char} {
2666      CHAR expected\n([pn] [$char tstr])
2667    }
2668    return [$char lower-case?]
2669  }
```

char->integer procedure

`char->integer` and `integer->char` convert between characters and their 16-bit numeric codes.

Example:

```
(char->integer #\A)   =>  65
```

char->integer (public)	
char	a character
Returns:	an integer

```
reg char->integer

proc ::constcl::char->integer {char} {
  return [MkNumber [scan [$char char] %c]]
}
```

integer->char procedure

Example:

```
(integer->char 97)   =>  #\a
```

integer->char (public)	
int	an integer
Returns:	a character

```
reg integer->char
```

```
proc ::constcl::integer->char {int} {
  if {$int == 10} {
    return [MkChar #\\newline]
  } elseif {$int == 32} {
    return [MkChar #\\space]
  } else {
    return [MkChar #\\[format %c [$int numval]]]
  }
}
```

char-upcase procedure

char-downcase procedure

`char-upcase` and `char-downcase` alter the case of a character.

Example:

```
(char-upcase #\a)   ==>  #\A
```

char-upcase, char-downcase (public)	
char	a character
Returns:	a character

```
reg char-upcase

proc ::constcl::char-upcase {char} {
  check {char? $char} {
    CHAR expected\n([pn] [$char tstr])
  }
  if {[$char char] in [::list " " "\n"]} {
    return $char
```

```
2694    } else {
2695      return [MkChar [
2696        ::string toupper [$char external]]]
2697    }
2698  }
```

```
2699  reg char-downcase
2700
2701  proc ::constcl::char-downcase {char} {
2702    check {char? $char} {
2703      CHAR expected\n([pn] [$char tstr])
2704    }
2705    if {[$char char] in [::list " " "\n"]} {
2706      return $char
2707    } else {
2708      return [MkChar [
2709        ::string tolower [$char external]]]
2710    }
2711  }
```

8.5 Control

This section concerns itself with procedures and the application of the same.

A `Procedure` object is a closure[18], storing the procedure's parameter list, the body, and the environment that is current when the object is created, i.e. when the procedure is defined (see page 80).

[18] See `https://en.wikipedia.org/wiki/Closure_(computer_programming)`

When a `Procedure` object is called, the body is evaluated in a new environment where the parameters are given values from the argument list and the outer link goes to the closure environment.

Procedure class

The Procedure class defines what capabilities a procedure has (in addition to those from the Base class), and also defines the internal representation of a procedure value expression. A procedure is stored in an instance as a tuple of formal parameters, body, and closed over environment. There is no method that yields the stored values.

```
2712 catch { ::constcl::Procedure destroy }
2713
2714 oo::class create ::constcl::Procedure {
2715   superclass ::constcl::Base
2716   variable parms body env
```

The `Procedure` constructor simply copies its arguments into the instance variables `parms`, `body`, and `env`.

Procedure constructor (internal)	
p	a Scheme formals list
b	an expression
e	an environment
Returns:	nothing

```
2717   constructor {p b e} {
2718     set parms $p
```

```
2719      set body $b
2720      set env $e
2721    }
2722  }
```

The `call` method makes each argument a tuple of `VARIABLE` and the argument value, storing the argument tuples in the list `vals`. Then an environment is created with the stored `parms`, `vals`, and stored `env` as arguments. The stored `body` is evaluated in this environment and the result is returned.

(Procedure instance) call (internal)	
args	some values
Returns:	a value

```
2723  oo::define ::constcl::Procedure method call {args} {
2724    set vals [lmap a $args {list VARIABLE $a}]
2725    ::constcl::eval $body [
2726      ::constcl::MkEnv $parms $vals $env]
2727  }
```

The `value` method is a dummy.

(Procedure instance) value (internal)	
Returns:	nothing

```
2728  oo::define ::constcl::Procedure method value {} {}
```

The `tstr` method yields the external representation of the procedure. It is used by error messages and by the `write` method.

(Procedure instance) tstr (internal)	
Returns:	a Tcl string

```
2729 oo::define ::constcl::Procedure method tstr {} {
2730   regexp {(\d+)} [self] -> num
2731   return "#<proc-$num>"
2732 }
```

MkProcedure generator

`MkProcedure` generates a Procedure object.

MkProcedure (internal)	
parms	a Scheme formals list
body	an expression
env	an environment
Returns:	a procedure

```
2733 interp alias {} ::constcl::MkProcedure \
2734   {} ::constcl::Procedure new
```

procedure? procedure

`procedure?` recognizes procedures either by type or by namespace, for procedures that are Tcl commands.

procedure? (public)	
val	a value
Returns:	a boolean

```
2735 reg procedure?
2736
2737 proc ::constcl::procedure? {val} {
```

```
2738    if {[typeof? $val Procedure] eq ${::#t}} {
2739      return ${::#t}
2740    } elseif {[::string match "::constcl::*" $val]} {
2741      return ${::#t}
2742    } else {
2743      return ${::#f}
2744    }
2745  }
```

apply procedure

`apply` applies a procedure to a Lisp list of Lisp arguments.
Example:

```
(apply + (list 2 3))   =>  5
```

apply (public)	
pr	a procedure
vals	a Lisp list of values
Returns:	what pr returns

```
2746  reg apply
2747
2748  proc ::constcl::apply {pr vals} {
2749    check {procedure? $pr} {
2750      PROCEDURE expected\n([pn] [$pr tstr] ...)
2751    }
2752    invoke $pr $vals
2753  }
```

map procedure

`map` iterates over one or more lists, taking an element from each list to pass to a procedure as an argument. The Lisp list of the results of the invocations is returned.

Example:

```
(map + '(1 2 3) '(5 6 7))   => (6 8 10)
```

map (public)	
pr	a procedure
args	some lists
Returns:	a Lisp list of values

```
reg map

proc ::constcl::map {pr args} {
  check {procedure? $pr} {
    PROCEDURE expected\n([pn] [$pr tstr] ...)
  }
```

The procedure iterates over the list of argument lists, converting each of them to a Tcl list.

```
  set arglists $args
  for {set i 0} \
    {$i < [llength $arglists]} \
    {incr i} {
    lset arglists $i [
      splitlist [lindex $arglists $i]]
  }
```

The procedure iterates over the items in each argument list (`item`) and each argument list (`arglist`), building a list of actual parameters (`actuals`). Then `pr` is invoked on the Lisp list of actuals and the result list-appended to `res`. After all the iterations, the Lisp list of items in `res` is returned.

```
2767    set res {}
2768    for {set item 0} \
2769      {$item < [llength [lindex $arglists 0]]} \
2770      {incr item} {
2771      set actuals {}
2772      for {set arglist 0} \
2773        {$arglist < [llength $arglists]} \
2774        {incr arglist} {
2775        lappend actuals [
2776          lindex $arglists $arglist $item]
2777      }
2778      lappend res [invoke $pr [list {*}$actuals]]
2779    }
2780    return [list {*}$res]
2781  }
```

for-each procedure

`for-each` iterates over one or more lists, taking an element from each list to pass to a procedure as an argument. The empty list is returned.

Example: (from R5RS; must be pasted as a oneliner for the `tkcon` repl to stomach it.)

```
(let ((v (make-vector 5)))
```

```
(for-each (lambda (i)
            (vector-set! v i (* i i)))
          '(0 1 2 3 4))
v)                          =>  #(0 1 4 9 16)
```

for-each (public)	
pr	a procedure
args	some lists
Returns:	the empty list

```
2782 reg for-each
2783
2784 proc ::constcl::for-each {proc args} {
2785   check {procedure? $proc} {
2786     PROCEDURE expected\n([pn] [$proc tstr] ...)
2787   }
```

The procedure iterates over the list of argument lists, converting each of them to a Tcl list.

```
2788   set arglists $args
2789   for {set i 0} \
2790     {$i < [llength $arglists]} \
2791     {incr i} {
2792     lset arglists $i [
2793       splitlist [lindex $arglists $i]]
2794   }
```

The procedure iterates over the items in each argument list (`item`) and each argument list (`arglist`), building a list of actual parameters (`actuals`). Then `proc` is invoked on the Lisp list of actuals. After all the iterations, the empty list is returned.

```
2795    for {set item 0} \
2796      {$item < [llength [lindex $arglists 0]]} \
2797      {incr item} {
2798      set actuals {}
2799      for {set arglist 0} \
2800        {$arglist < [llength $arglists]} \
2801        {incr arglist} {
2802        lappend actuals [
2803          lindex $arglists $arglist $item]
2804      }
2805      invoke $proc [list {*}$actuals]
2806    }
2807    return ${::#NIL}
2808  }
```

8.6 Input and output

Like most programming languages, Scheme has input and output facilities beyond mere `read` and `write`. I/O is based on the *port* abstraction (see page 31) of a character supplying or receiving device. There are four kinds of ports:

1. file input (InputPort)
2. file output (OutputPort)
3. string input (StringInputPort)
4. string output (StringOutputPort)

and there is also the `Port` kind, which isn't used other than as a base class.

Port class

```
2809 oo::abstract create ::constcl::Port {
2810   superclass ::constcl::Base
2811   variable handle
```

The `Port` constructor uses a fake argument to store a value in the instance variable `handle`. If the value isn't provided, `handle` gets the value of the empty list.

Port constructor (internal)	
?h?	a channel handle
Returns:	nothing

```
2812   constructor {args} {
2813     if {[llength $args]} {
2814       lassign $args handle
2815     } else {
2816       set handle ${::#NIL}
2817     }
2818   }
```

The `handle` method yields the stored handle value.

(concrete instance) handle (internal)	
Returns:	a channel handle or NIL

```
2819   method handle {} {
2820     set handle
2821   }
```

The `close` method acts to close the stored handle's channel, and sets the stored handle to the empty list.

(concrete instance) close (internal)	
Returns:	nothing

```
2822    method close {} {
2823      close $handle
2824      set handle ${::#NIL}
2825      return
2826    }
2827  }
```

InputPort class

The InputPort class extends Port with the ability to open a channel for reading, and to get a character from the channel and to detect end-of-file.

```
2828  oo::class create ::constcl::InputPort {
2829    superclass ::constcl::Port
2830    variable handle
```

The InputPort `open` method takes a file name and attempts to open it for reading. If it succeeds, it sets the stored handle to the opened channel. If it fails, it sets the stored handle to the empty list.

(InputPort instance) open (internal)	
name	a filename string
Returns:	a channel handle or NIL

```
2831    method open {name} {
2832      try {
2833        set handle [open [$name value] "r"]
2834      } on error {} {
2835        set handle ${::#NIL}
2836      }
2837      return $handle
2838    }
```

The `get` method reads one character from the channel of the stored handle.

(InputPort instance) get (internal)	
Returns:	a Tcl character

```
2839    method get {} {
2840      chan read $handle 1
2841    }
```

The `eof` method reports end-of-file status on the channel of the stored handle.

(InputPort instance) eof (internal)	
Returns:	a Tcl truth value (1 or 0)

```
2842    method eof {} {
2843      chan eof $handle
2844    }
```

The `copy` method returns a new instance of InputPort which is a copy of this instance, sharing the stored handle.

(InputPort instance) copy (internal)	
Returns:	an input port

```
2845    method copy {} {
2846      ::constcl::InputPort new $handle
2847    }
```

The `tstr` method yields the external representation of the port as a Tcl string. It is used by error messages and the `write` method.

(InputPort instance) tstr (internal)	
Returns:	a Tcl string

```
2848    method tstr {} {
2849      regexp {(\d+)} [self] -> num
2850      return "#<input-port-$num>"
2851    }
2852  }
```

MkInputPort generator

`MkInputPort` generates an InputPort object.

MkInputPort (internal)	
?handle?	a channel handle
Returns:	an input port

```
2853  interp alias {} ::constcl::MkInputPort \
2854    {} ::constcl::InputPort new
```

StringInputPort class

The StringInputPort class extends Port with the ability to get a character from the buffer and detect end-of-file. It turns `open` and `close` into no-op methods.

```
2855 oo::class create ::constcl::StringInputPort {
2856   superclass ::constcl::Port
2857   variable buffer read_eof
```

The StringInputPort constructor simply copies a given string into the stored buffer and sets the `read_eof` state variable to 0.

StringInputPort constructor (internal)	
str	a Tcl string
Returns:	nothing

```
2858   constructor {str} {
2859     set buffer $str
2860     set read_eof 0
2861   }
```

The `open` and `close` methods are present but don't do anything.

(StringInputPort instance) open (internal)	
name	a filename string
Returns:	nothing

(StringInputPort instance) close (internal)	
Returns:	nothing

```
2862   method open {name} {}
2863   method close {} {}
```

The StringInputPort `get` method reads one character from the buffer. If the buffer is empty, `#EOF` is returned and `read_eof` is set to 1. The buffer is reduced by one character.

(StringInputPort instance) get (internal)	
Returns:	a Tcl character or end of file

```
2864  method get {} {
2865    if {[::string length $buffer] == 0} {
2866      set read_eof 1
2867      return #EOF
2868    }
2869    set c [::string index $buffer 0]
2870    set buffer [::string range $buffer 1 end]
2871    return $c
2872  }
```

The `eof` method reports end-of-file status on the buffer.

(StringInputPort instance) eof (internal)	
Returns:	a Tcl truth value (1 or 0)

```
2873  method eof {} {
2874    return $read_eof
2875  }
```

The `copy` method creates a new instance with a (non-shared) copy of the buffer such as it is at this point in time.

(StringInputPort instance) copy (internal)	
Returns:	a string input port

```
2876  method copy {} {
2877    ::constcl::StringInputPort new $buffer
2878  }
```

The `tstr` method yields the external representation of the string input port as a Tcl string. It is used by error messages and the `write` method.

(StringInputPort instance) tstr (internal)	
Returns:	a Tcl string

```
2879    method tstr {} {
2880      regexp {(\d+)} [self] -> num
2881      return "#<string-input-port-$num>"
2882    }
2883  }
```

MkStringInputPort generator

`MkStringInputPort` generates a StringInputPort object.

MkStringInputPort (internal)	
str	a string
Returns:	a string input port

```
2884  interp alias {} ::constcl::MkStringInputPort \
2885    {} ::constcl::StringInputPort new
```

OutputPort class

OutputPort extends Port with the ability to open a channel for writing, and to put a string through the channel, print a newline, and flush the channel.

```
2886 oo::class create ::constcl::OutputPort {
2887   superclass ::constcl::Port
2888   variable handle
```

The OutputPort `open` method attempts to open a channel for writing on a given file name, setting the stored handle to the channel if it succeeds and to the empty list if it fails.

The `open` method is locked with an error command for safety. Only remove this line if you really know what you're doing: once it is unlocked, the `open` method can potentially overwrite existing files.

(OutputPort instance) open (internal)	
name	a filename string
Returns:	a channel handle or NIL

```
2889   method open {name} {
2890     ::error "remove this line to use"
2891     try {
2892       set handle [open [$name value] "w"]
2893     } on error {} {
2894       set handle ${::#NIL}
2895     }
2896     return $handle
2897   }
```

The OutputPort `put` method outputs a string on the channel in the stored handle.

(OutputPort instance) put (internal)	
str	a Tcl string
Returns:	nothing

```
2898    method put {str} {
2899      puts -nonewline $handle $str
2900    }
```

The `newline` method prints a newline on the channel in the stored handle.

(OutputPort instance) newline (internal)	
Returns:	nothing

```
2901    method newline {} {
2902      puts $handle {}
2903    }
```

The `flush` method flushes the output channel in the stored handle.

(OutputPort instance) flush (internal)	
Returns:	nothing

```
2904    method flush {} {
2905      flush $handle
2906    }
```

The `copy` method returns a new instance of OutputPort which is a copy of this instance, sharing the stored handle.

```
2907    method copy {} {
2908      ::constcl::OutputPort new $handle
2909    }
```

The `tstr` method yields the external representation of the port as a Tcl string. It is used by error messages and the `write` method.

(OutputPort instance) tstr (internal)	
Returns:	a Tcl string

```
2910    method tstr {} {
2911      regexp {(\d+)} [self] -> num
2912      return "#<output-port-$num>"
2913    }
2914  }
```

MkOutputPort generator

`MkOutputPort` generates an OutputPort object.

MkOutputPort (internal)	
?handle?	a channel handle
Returns:	an output port

```
2915  interp alias {} ::constcl::MkOutputPort \
2916    {} ::constcl::OutputPort new
```

StringOutputPort class

StringOutputPort extends Port with the ability to put strings into a string buffer. The `tostring` method yields the current contents of the buffer.

```
2917 oo::class create ::constcl::StringOutputPort {
2918   superclass ::constcl::Port
2919   variable buffer
```

The StringOutputPort constructor uses a fake argument to optionally initialize the internal buffer, which otherwise is empty.

StringOutputPort constructor (internal)	
?str?	a Tcl string
Returns:	nothing

```
2920   constructor {args} {
2921     if {[llength $args]} {
2922       lassign $args str
2923       set buffer $str
2924     } else {
2925       set buffer {}
2926     }
2927   }
```

The `open` and `close` methods are present but don't do anything.

(StringOutputPort instance) open (internal)	
name	a filename string
Returns:	nothing

(StringOutputPort instance) close (internal)	
Returns:	nothing

```
2928   method open {name} {}
2929   method close {} {}
```

The StringOutputPort `put` method appends a string to the internal buffer.

(StringOutputPort instance) put (internal)	
str	a Tcl string
Returns:	nothing

```
2930  method put {str} {
2931    append buffer $str
2932    return
2933  }
```

The `newline` method appends a newline character to the internal buffer.

(StringOutputPort instance) newline (internal)	
Returns:	nothing

```
2934  method newline {} {
2935    append buffer \n
2936  }
```

The `flush` method is present but does nothing.

(StringOutputPort instance) flush (internal)	
Returns:	nothing

```
2937  method flush {} {}
```

The `tostring` method dumps the internal buffer as a string.

(StringOutputPort instance) tostring (internal)	
Returns:	a string

```
2938    method tostring {} {
2939      MkString $buffer
2940    }
```

The copy method returns a new instance of StringOutputPort which is a copy of this instance, with a (non-shared) copy of the internal buffer such as it is at this point in time.

(StringOutputPort instance) copy (internal)	
Returns:	a string output port

```
2941    method copy {} {
2942      ::constcl::StringOutputPort new $buffer
2943    }
```

The tstr method yields the external representation of the string output port as a Tcl string. It is used by error messages and the write method.

(StringOutputPort instance) tstr (internal)	
Returns:	a Tcl string

```
2944    method tstr {} {
2945      regexp {(\d+)} [self] -> num
2946      return "#<string-output-port-$num>"
2947    }
2948  }
```

MkStringOutputPort generator

MkStringOutputPort generates a StringOutputPort object.

MkStringOutputPort (internal)	
?str?	a string
Returns:	a string output port

```
interp alias {} ::constcl::MkStringOutputPort \
  {} ::constcl::StringOutputPort new
```

Input_Port variable
Output_port variable
These two variables store the current configuration of the shared input and output ports globally. They are initially set to standard input and output respectively.

```
set ::constcl::Input_port [
  ::constcl::MkInputPort stdin]
set ::constcl::Output_port [
  ::constcl::MkOutputPort stdout]
```

port? procedure

`port?` recognizes Port objects, i.e. all kinds of ports.

```
reg port?

proc ::constcl::port? {val} {
  typeof? $val Port
}
```

call-with-input-file procedure

`call-with-input-file` opens a file for input and passes the port to `proc`. The file is closed again once `proc` returns. The result of the call is returned.

call-with-input-file (public)	
filename	a filename string
proc	a procedure
Returns:	a value

```
2960 reg call-with-input-file
2961
2962 proc ::constcl::call-with-input-file {filename proc} {
2963   set port [open-input-file $filename]
2964   set res [invoke $proc [list $port]]
2965   close-input-port $port
2966   $port destroy
2967   return $res
2968 }
```

call-with-output-file procedure

`call-with-output-file` opens a file for output and passes the port to `proc`. The file is closed again once `proc` returns. The result of the call is returned.

call-with-output-file (public)	
filename	a filename string
proc	a procedure
Returns:	a value

```
reg call-with-output-file

proc ::constcl::call-with-output-file {filename proc} {
  set port [open-output-file $filename]
  set res [invoke $proc [list $port]]
  close-output-port $port
  $port destroy
  return $res
}
```

input-port? procedure

`input-port?` recognizes an `InputPort` or `StringInputPort` object.

input-port? (public)	
val	a value
Returns:	a boolean

```
reg input-port?

proc ::constcl::input-port? {val} {
  if {[T typeof? $val InputPort]} {
    return ${::#t}
  } elseif {[T typeof? $val StringInputPort]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

output-port? procedure

`output-port?` recognizes an `OutputPort` or `StringOutputPort` object.

output-port? (public)	
val	a value
Returns:	a boolean

```
reg output-port?

proc ::constcl::output-port? {val} {
  if {[T typeof? $val OutputPort]} {
    return ${::#t}
  } elseif {[T typeof? $val StringOutputPort]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

current-input-port procedure

`current-input-port` makes a copy of the current shared input port.

current-input-port (public)	
Returns:	a port

```
reg current-input-port

proc ::constcl::current-input-port {} {
  return [$::constcl::Input_port copy]
}
```

current-output-port procedure

`current-output-port` makes a copy of the current shared output port.

current-output-port (public)	
Returns:	a port

```
reg current-output-port

proc ::constcl::current-output-port {} {
  return [$::constcl::Output_port copy]
}
```

with-input-from-file procedure

`with-input-from-file` opens a file for input and calls a 'thunk' while the file is open. The file is closed again when the call is done.

with-input-from-file (public)	
filename	a filename string
thunk	a procedure
Returns:	nothing

```
reg with-input-from-file

proc ::constcl::with-input-from-file {filename thunk} {
  set newport [open-input-file $filename]
  if {[$newport handle] ne ${::#NIL}} {
    set oldport $::constcl::Input_port
    set ::constcl::Input_port $newport
    $thunk call
```

```
3018      set ::constcl::Input_port $oldport
3019      close-input-port $newport
3020    }
3021    $newport destroy
3022  }
```

with-output-to-file procedure

`with-output-to-file` opens a file for output and calls a 'thunk' while the file is open. The file is closed again when the call is done.

with-output-to-file (public)	
filename	a filename string
thunk	a procedure
Returns:	nothing

```
3023  reg with-output-to-file
3024
3025  proc ::constcl::with-output-to-file {filename thunk} {
3026    set newport [open-output-file $filename]
3027    if {[$newport handle] ne ${::#NIL}} {
3028      set oldport $::constcl::Output_port
3029      set ::constcl::Output_port $newport
3030      $thunk call
3031      set ::constcl::Output_port $oldport
3032      close-input-port $newport
3033    }
3034    $newport destroy
3035  }
```

open-input-file procedure

`open-input-file` opens a file for input and returns the port.

open-input-file (public)	
filename	a filename string
Returns:	an input port

```
reg open-input-file

proc cnof {} {return "could not open file"}
proc fae {} {return "file already exists"}

proc ::constcl::open-input-file {filename} {
  set p [MkInputPort]
  $p open $filename
  if {[$p handle] eq ${::#NIL}} {
    set fn [$filename value]
    error "open-input-file: [cnof] $fn"
  }
  return $p
}
```

open-output-file procedure

`open-output-file` opens a file for output and returns the port. Throws an error if the file already exists.

open-output-file (public)	
filename	a filename string
Returns:	an output port

```
reg open-output-file
```

```
3051
3052 proc ::constcl::open-output-file {filename} {
3053   if {[file exists $filename]} {
3054     error "open-output-file: [fae] $filename"
3055   }
3056   set p [MkOutputPort]
3057   $p open $filename
3058   if {[$p handle] eq ${::#NIL}} {
3059     error "open-output-file: [cnof] $filename"
3060   }
3061   return $p
3062 }
```

close-input-port procedure

`close-input-port` closes an input port.

close-input-port (public)	
port	an input port
Returns:	nothing

```
3063 reg close-input-port
3064
3065 proc ::constcl::close-input-port {port} {
3066   if {[$port handle] eq "stdin"} {
3067     error "don't close the standard input port"
3068   }
3069   $port close
3070 }
```

close-output-port procedure

`close-output-port` closes an output port.

close-output-port (public)	
port	an output port
Returns:	nothing

```
reg close-output-port

proc ::constcl::close-output-port {port} {
  if {[$port handle] eq "stdout"} {
    error "don't close the standard output port"
  }
  $port close
}
```

`write` is implemented in the output (see page 125) chapter. `display` is implemented in the same chapter.

newline procedure

`newline` outputs a newline character. Especially helpful when using `display` for output, since it doesn't end lines with newline.

newline (public)	
?port?	an output port
Returns:	nothing

```
reg newline

proc ::constcl::newline {args} {
  if {[llength $args]} {
```

```
3083      lassign $args port
3084    } else {
3085      set port [current-output-port]
3086    }
3087    $port newline
3088  }
```

---

load

`load` reads a Scheme source file and evals the expressions in it in the global environment. The procedure is a ConsTcl mix of Scheme calls and Tcl syntax.

load (public)	
filename	a filename string
Returns:	nothing

```
3089  reg load
3090
3091  proc ::constcl::load {filename} {
3092    try {
3093      open-input-file $filename
3094    } on ok port {
3095    } on error {} {
3096      return
3097    }
3098    if {[$port handle] ne ${::#NIL}} {
3099      set expr [read $port]
3100      while {$expr ne "#EOF"} {
3101        eval $expr
3102        set expr [read $port]
3103      }
3104      close-input-port $port
3105    }
3106    $port destroy
```

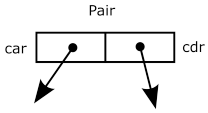
```
3107 }
```

8.7 Pairs and lists

List processing is another of Lisp's great strengths. In Lisp, lists (which are actually tree structures) are composed of *pairs*, which in the most elementary case are constructed using calls to the `cons` function. Example:

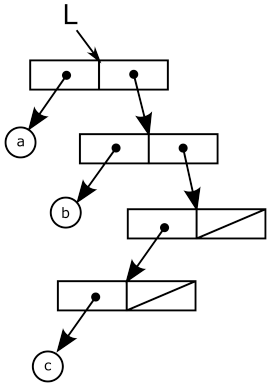
```
(cons 'a
  (cons 'b (cons (cons 'c '()) '())))  ==> (a b (c))
```

A *pair* consists of a pair of pointers, named the *car* and the *cdr*[19].



The example above would look like this (we'll name it L). `car L` is the symbol `a`, and `cdr L` is the list `(b (c))`. `cadr L` (the `car` of `cdr L`) is `b`.

[19]There are historical, not very interesting, reasons for this naming.



All program source code has a tree structure, even though this is usually mostly hidden by the language. Lisp, on the other hand, makes the tree structure fully explicit by using the same notation for source code as for list data (hence all the parentheses).

Pair class

The Pair class defines what capabilities a pair has (in addition to those from the Base class), and also defines the internal representation of a pair value expression. A pair is stored in an instance as a couple of pointers, and the `car` and `cdr` methods yield each of them as result.

```
3108 oo::class create ::constcl::Pair {
3109   superclass ::constcl::Base
3110   variable car cdr constant
```

The constructor stores values into the `car` and `cdr` variables, and sets `constant` to 0, denoting that the pair is mutable.

Pair constructor (internal)	
a	a value
d	a value
Returns:	nothing

```
3111  constructor {a d} {
3112    set car $a
3113    set cdr $d
3114    set constant 0
3115  }
```

The `value` method is a synonym for `tstr`.

(Pair instance) value (internal)	
Returns:	an external representation of a pair

```
3116  method value {} {
3117    my tstr
3118  }
```

The `car` method returns the value stored in the `car` variable.

(Pair instance) car (internal)	
Returns:	a value

```
3119  method car {} {
3120    set car
3121  }
```

The `cdr` method returns the value stored in the `cdr` variable.

(Pair instance) cdr (internal)	
Returns:	a value

```
3122  method cdr {} {
3123    set cdr
3124  }
```

The `set-car!` method modifies the value stored in the `car` variable.

(Pair instance) set-car! (internal)	
val	a value
Returns:	a pair

```
3125  method set-car! {val} {
3126    ::constcl::check {my mutable?} {
3127      Can't modify a constant pair
3128    }
3129    set car $val
3130    self
3131  }
```

The `set-cdr!` method modifies the value stored in the `cdr` variable.

(Pair instance) set-cdr! (internal)	
val	a value
Returns:	a pair

```
3132  method set-cdr! {val} {
3133    ::constcl::check {my mutable?} {
3134      Can't modify a constant pair
3135    }
```

```
3136      set cdr $val
3137      self
3138    }
```

The `mkconstant` method changes the instance from mutable to immutable.

(Pair instance) mkconstant (internal)	
Returns:	nothing

```
3139    method mkconstant {} {
3140      set constant 1
3141      return
3142    }
```

The `constant` method signals whether the pair instance is immutable.

(Pair instance) constant (internal)	
Returns:	a Tcl truth value (1 or 0)

```
3143    method constant {} {
3144      return $constant
3145    }
```

The `mutable?` method is a predicate that tells if the pair instance is mutable or not.

(Pair instance) mutable? (internal)	
Returns:	a boolean

```
3146    method mutable? {} {
3147      expr {$constant ? ${::#f} : ${::#t}}
3148    }
```

The `write` method prints an external representation of the pair on the given port.

(Pair instance) write (internal)	
port	an output port
Returns:	nothing

```
3149    method write {port} {
3150      $port put "("
3151      ::constcl::write-pair $port [self]
3152      $port put ")"
3153    }
```

The `tstr` method yields the external representation of the pair instance as a Tcl string. It is used by error messages.

(Pair instance) tstr (internal)	
Returns:	an external representation of a pair

```
3154    method tstr {} {
3155      format "(%s)" [::constcl::tstr-pair [self]]
3156    }
3157  }
```

MkPair generator

`MkPair` generates a Pair object. Shorter form: `cons`.

MkPair (internal)	
car	a value
cdr	a value
Returns:	a pair

```
interp alias {} ::constcl::MkPair \
  {} ::constcl::Pair new
```

pair? procedure

pair? (public)	
val	a value
Returns:	a boolean

```
reg pair?

proc ::constcl::pair? {val} {
  typeof? $val Pair
}
```

tstr-pair procedure

Helper procedure to make a string representation of a list.

tstr-pair (internal)	
pair	a pair
Returns:	a Tcl string

```
proc ::constcl::tstr-pair {pair} {
  # take a pair and make a string of the car
  # and the cdr of the stored value
  set str {}
  set a [car $pair]
  set d [cdr $pair]
  ::append str [$a tstr]
```

```
3172    if {[T [pair? $d]]} {
3173      # cdr is a cons pair
3174      ::append str " "
3175      ::append str [tstr-pair $d]
3176    } elseif {[T [null? $d]]} {
3177      # cdr is nil
3178      return $str
3179    } else {
3180      # it is an atom
3181      ::append str " . "
3182      ::append str [$d tstr]
3183    }
3184    return $str
3185  }
```

cons procedure

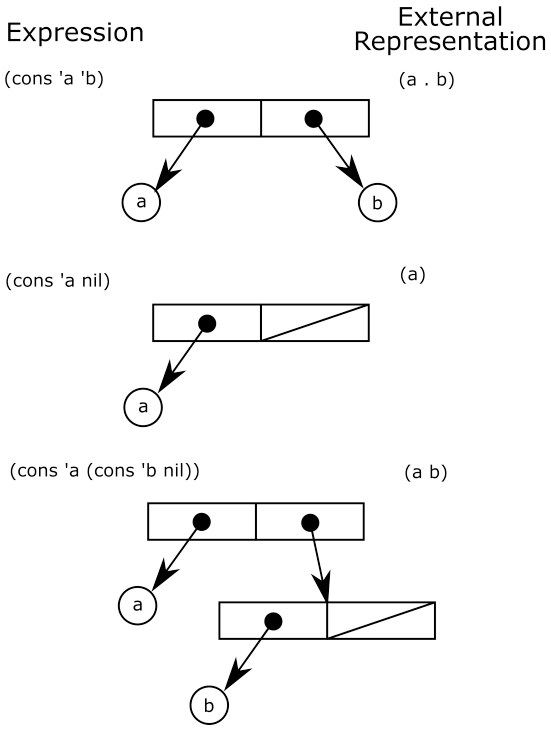
`cons` joins two values in a pair; useful in many operations such as pushing a new value onto a list.

Example:

```
(cons 'a 'b)              ==>  (a . b)
(cons 'a nil)             ==>  (a)
(cons 'a (cons 'b nil))   ==>  (a b)
```

cons (public)	
car	a value
cdr	a value
Returns:	a pair

```
3186  reg cons
```



Examples of consing

```
proc ::constcl::cons {car cdr} {
```

```
3189    MkPair $car $cdr
3190  }
```

car procedure

`car` gets the contents of the first cell in a pair.
Example:

```
(car '(a b))   ==>  a
```

car (public)	
pair	a pair
Returns:	a value

```
3191  reg car
3192
3193  proc ::constcl::car {pair} {
3194    $pair car
3195  }
```

cdr procedure

`cdr` gets the contents of the second cell in a pair.
Example:

```
(cdr '(a b))   ==>  (b)
```

cdr (public)	
pair	a pair
Returns:	a value

```
reg cdr

proc ::constcl::cdr {pair} {
  $pair cdr
}
```

caar to **cddddr**

`car` and `cdr` can be combined to form 28 composite access operations.

```
foreach ads {
  aa
  ad
  da
  dd
  aaa
  ada
  daa
  dda
  aad
  add
  dad
  ddd
  aaaa
  adaa
  daaa
  ddaa
  aada
  adda
  dada
  ddda
  aaad
  adad
  daad
```

```
   ddad
   aadd
   addd
   dadd
   dddd
} {
    reg c${ads}r

    proc ::constcl::c${ads}r {x} "
        foreach c \[lreverse \[split $ads {}\]\] {
            if {\$c eq \"a\"} {
                set x \[car \$x\]
            } else {
                set x \[cdr \$x\]
            }
        }
        return \$x
    "

}
```

set-car! procedure

`set-car!` sets the contents of the first cell in a pair.

Example:

```
(let ((pair (cons 'a 'b)) (val 'x))
  (set-car! pair val))                ==>  (x . b)
```

set-car! (public)	
pair	a pair
val	a value
Returns:	a pair

```
reg set-car!

proc ::constcl::set-car! {pair val} {
  $pair set-car! $val
}
```

set-cdr! procedure

`set-cdr!` sets the contents of the second cell in a pair.

Example:

```
(let ((pair (cons 'a 'b)) (val 'x))
  (set-cdr! pair val))                ==>  (a . x)
```

set-cdr! (public)	
pair	a pair
val	a value
Returns:	a pair

```
reg set-cdr!

proc ::constcl::set-cdr! {pair val} {
  $pair set-cdr! $val
}
```

list? procedure

The `list?` predicate tests if a pair is part of a proper list, one that ends with NIL. See the figure showing proper and improper lists (see page 43).

list? (public)	
val	a value
Returns:	a boolean

```
reg list?

proc ::constcl::list? {val} {
  set visited {}
  if {[T [null? $val]]} {
      return ${::#t}
  } elseif {[T [pair? $val]]} {
      return [listp $val]
  } else {
      return ${::#f}
  }
}
```

listp procedure

`listp` is a helper procedure that recursively traverses a pair trail to find out if it is cyclic or ends in an atom, which means that the procedure returns false, or if it ends in `NIL`, which means that it returns true.

listp (internal)	
pair	a pair
Returns:	a boolean

```
proc ::constcl::listp {pair} {
```

```
3268    upvar visited visited
3269    if {$pair in $visited} {
3270      return ${::#f}
3271    }
3272    lappend visited $pair
3273    if {[T [null? $pair]]} {
3274      return ${::#t}
3275    } elseif {[T [pair? $pair]]} {
3276      return [listp [cdr $pair]]
3277    } else {
3278      return ${::#f}
3279    }
3280  }
```

list procedure

`list` constructs a Lisp list from a number of values.
Example:

```
(list 1 2 3)   ==>  (1 2 3)
```

list (public)	
args	some values
Returns:	a Lisp list of values

```
3281  reg list
3282
3283  proc ::constcl::list {args} {
3284    if {[llength $args] == 0} {
3285      return ${::#NIL}
3286    } else {
3287      set prev ${::#NIL}
```

```
3288      foreach obj [lreverse $args] {
3289        set prev [cons $obj $prev]
3290      }
3291      return $prev
3292    }
3293  }
```

length procedure

`length` reports the length of a Lisp list.

Example:

```
(length '(a b c d))   ==>  4
```

length (public)	
pair	a pair
Returns:	a number

```
3294  reg length
3295
3296  proc ::constcl::length {pair} {
3297    check {list? $pair} {
3298      LIST expected\n([pn] lst)
3299    }
3300    MkNumber [length-helper $pair]
3301  }
```

length-helper procedure

`length-helper` is a helper procedure which measures a list recursively.

length-helper (internal)	
pair	a pair
Returns:	a Tcl number

```
3302 proc ::constcl::length-helper {pair} {
3303   if {[T [null? $pair]]} {
3304     return 0
3305   } else {
3306     return [expr {1 +
3307       [length-helper [cdr $pair]]}]
3308   }
3309 }
```

append procedure

append joins lists together.

Example:

```
(append '(a b) '(c d))   ==>  (a b c d)
```

append (public)	
args	some lists
Returns:	a Lisp list of values

```
3310 reg append
3311
3312 proc ::constcl::append {args} {
3313   set prev [lindex $args end]
3314   foreach r [lreverse [lrange $args 0 end-1]] {
3315     check {list? $r} {
3316       LIST expected\n([pn] [$r tstr])
```

```
3317      }
3318      set prev [copy-list $r $prev]
3319    }
3320    set prev
3321  }
```

copy-list procedure

`copy-list` joins together two lists by recursively consing items from the first list towards the second.

copy-list (internal)	
pair	a pair
next	a Lisp list of values
Returns:	a Lisp list of values

```
3322  proc ::constcl::copy-list {pair next} {
3323    if {[T [null? $pair]]} {
3324      set next
3325    } elseif {[T [null? [cdr $pair]]]} {
3326      cons [car $pair] $next
3327    } else {
3328      cons [car $pair] [copy-list [cdr $pair] $next]
3329    }
3330  }
```

reverse procedure

`reverse` produces a reversed copy of a Lisp list.

Example:

```
(reverse '(a b c))   ==>  (c b a)
```

reverse (public)	
vals	a Lisp list of values
Returns:	a Lisp list of values

```
reg reverse

proc ::constcl::reverse {vals} {
  list {*}[lreverse [splitlist $vals]]
}
```

list-tail procedure

Given a list index, `list-tail` yields the sublist starting from that index.

Example:

```
(let ((lst '(a b c d e f)) (k 3))
  (list-tail lst k))                ==>  (d e f)
```

list-tail (public)	
vals	a Lisp list of values
k	a number
Returns:	a Lisp list of values

```
reg list-tail

proc ::constcl::list-tail {vals k} {
  if {[T [zero? $k]]} {
    return $vals
  } else {
    list-tail [cdr $vals] [- $k [N 1]]
```

```
  }
}
```

list-ref procedure

`list-ref` yields the list item at a given index (0-based).

Example:

```
(let ((lst '(a b c d e f)) (k 3))
  (list-ref lst k))                    ==>  d
```

list-ref (public)	
vals	a Lisp list of values
k	a number
Returns:	a value

```
reg list-ref

proc ::constcl::list-ref {vals k} {
  car [list-tail $vals $k]
}
```

memq procedure

memv procedure
member procedure

`memq`, `memv`, and `member` return the sublist starting with a given item, or `#f` if there is none. They use `eq?`, `eqv?`, and `equal?`, respectively, for the comparison.

Example:

```
(let ((lst '(a b c d e f)) (val 'd))
  (memq val lst))                          ==>  (d e f)
```

memq (public)	
val1	a value
val2	a Lisp list of values
Returns:	a Lisp list of values OR #f

```
reg memq

proc ::constcl::memq {val1 val2} {
  return [member-proc eq? $val1 $val2]
}
```

memv (public)	
val1	a value
val2	a Lisp list of values
Returns:	a Lisp list of values OR #f

```
reg memv

proc ::constcl::memv {val1 val2} {
  return [member-proc eqv? $val1 $val2]
}
```

member (public)	
val1	a value
val2	a Lisp list of values
Returns:	a Lisp list of values OR #f

```
reg member

proc ::constcl::member {val1 val2} {
  return [member-proc equal? $val1 $val2]
}
```

member-proc procedure

The `member-proc` helper procedure does the work for the `memq`, `memv`, and `member` procedures. It works by comparing against the `car` of the list, then recursively taking the `cdr` of the list.

member-proc (internal)	
epred	an equivalence predicate
val1	a value
val2	a Lisp list of values
Returns:	a Lisp list of values OR #f

```
proc ::constcl::member-proc {epred val1 val2} {
  switch $epred {
    eq? { set name "memq" }
    eqv? { set name "memv" }
    equal? { set name "member" }
  }
  check {list? $val2} {
    LIST expected\n($name [$val1 tstr] [$val2 tstr])
  }
  if {[T [null? $val2]]} {
    return ${::#f}
  } elseif {[T [pair? $val2]]} {
    if {[T [$epred $val1 [car $val2]]]} {
      return $val2
    } else {
```

```
3380        return [member-proc $epred $val1 [cdr $val2]]
3381      }
3382    }
3383  }
```

assq procedure

assv procedure
assoc procedure

`assq`, `assv`, and `assoc` scan an association list and return the association pair with a given key, or `#f` if there is none. They use `eq?`, `eqv?`, and `equal?`, respectively, for the comparison. They implement lookup in the kind of lookup table known as an association list, or *alist*.

Example:

```
(let ((e '((a . 1) (b . 2) (c . 3)))
      (key 'a))
  (assq key e))                          ==> (a . 1)
```

assq (public)	
val1	a value
val2	an association list
Returns:	an association pair or #f

```
3384  reg assq
3385
3386  proc ::constcl::assq {val1 val2} {
3387    return [assoc-proc eq? $val1 $val2]
3388  }
```

assv (public)	
val1	a value
val2	an association list
Returns:	an association pair or #f

```
reg assv

proc ::constcl::assv {val1 val2} {
  return [assoc-proc eqv? $val1 $val2]
}
```

assoc (public)	
val1	a value
val2	an association list
Returns:	an association pair or #f

```
reg assoc

proc ::constcl::assoc {val1 val2} {
  return [assoc-proc equal? $val1 $val2]
}
```

assoc-proc procedure

`assoc-proc` is a helper procedure which does the work for `assq`, `assv`, and `assoc`.

assoc-proc (internal)	
epred	an equivalence predicate
val1	a value
val2	an association list
Returns:	an association pair or #f

```
3399 proc ::constcl::assoc-proc {epred val1 val2} {
3400   switch $epred {
3401     eq? { set name "assq" }
3402     eqv? { set name "assv" }
3403     equal? { set name "assoc" }
3404   }
3405   check {list? $val2} {
3406     LIST expected\n($name [$val1 tstr] [$val2 tstr])
3407   }
3408   if {[T [null? $val2]]} {
3409     return ${::#f}
3410   } elseif {[T [pair? $val2]]} {
3411     if {[T [pair? [car $val2]]] &&
3412       [T [$epred $val1 [caar $val2]]]} {
3413       return [car $val2]
3414     } else {
3415       return [assoc-proc $epred $val1 [cdr $val2]]
3416     }
3417   }
3418 }
```

8.8 Strings

Strings are sequences of characters. They are the most common form of real-world data in computing nowadays, having outpaced numbers some time ago. Lisp has strings, both constant and mutable, but some of the uses for strings in other languages are instead taken up by symbols.

String class

Strings have the internal representation of a vector of character objects, with the data elements of 1) the vector address of the first element, and 2) the length of the vector. The external representation of a string is enclosed within double quotes, with double quotes and backslashes within the string escaped with a backslash.

As an extension, a `\n` pair in the external representation is stored as a newline character. It is restored to `\n` if the string is printed using `write`, but remains a newline character if the string is printed using `display`.

```
3419 oo::class create ::constcl::String {
3420   superclass ::constcl::Base
3421   variable data constant
```

The String constructor converts the given string to print form (no escaping backslashes), sets the string length, and allocates that much vector memory to store the string. The characters that make up the string are stored as Char objects. Finally the string's data tuple (a pair holding the address to the first stored character and the length of the string) is stored and `constant` is set to 0, indicating a mutable string.

String constructor (internal)	
val	an external repr. of a string, w/o double quotes
Returns:	nothing

```
3422   constructor {val} {
```

```
3423      set val [string map {\\\\ \\ \\\" \" \\n \n} $val]
3424      set len [::string length $val]
3425      # allocate vector space for the string's
3426      # characters
3427      set vsa [::constcl::vsAlloc $len]
3428      # store the characters in vector space, as
3429      # Char objects
3430      set idx $vsa
3431      foreach elt [split $val {}] {
3432        if {$elt eq " "} {
3433          set c #\\space
3434        } elseif {$elt eq "\n"} {
3435          set c #\\newline
3436        } else {
3437          set c #\\$elt
3438        }
3439        lset ::constcl::vectorSpace $idx \
3440          [::constcl::MkChar $c]
3441        incr idx
3442      }
3443      # store the basic vector data: address of
3444      # first character and length
3445      set data [
3446        ::constcl::cons [N $vsa] [N $len]]
3447      set constant 0
3448    }
```

The = method tells if the stored string is equal to a given string.

(String instance) = (internal)	
str	a string
Returns:	a Tcl truth value (1 or 0)

```
3449    method = {str} {
```

```
3450      ::string equal [my value] [$str value]
3451    }
```

The `cmp` method compares the stored string to a given string. Returns -1, 0, or 1, depending on whether the stored string is less than, equal to, or greater than the other string.

(String instance) cmp (internal)	
str	a string
Returns:	a comparison value: -1, 0, or 1

```
3452    method cmp {str} {
3453      ::string compare [my value] [$str value]
3454    }
```

The `length` method returns the length (as a Number object) of the internal representation of the string in characters.

(String instance) length (internal)	
Returns:	a number

```
3455    method length {} {
3456      ::constcl::cdr $data
3457    }
```

The `ref` method, given an index value which is between 0 and the length of the string, returns the character at that index position.

(String instance) ref (internal)	
k	a number
Returns:	a character

```
3458  method ref {k} {
3459    set k [$k numval]
3460    if {$k < 0 || $k >= [[my length] numval]} {
3461      ::error "index out of range\n$k"
3462    }
3463    lindex [my store] $k
3464  }
```

The `store` method presents the range in vector memory where the string is stored.

(String instance) store (internal)	
Returns:	a Tcl list of characters

```
3465  method store {} {
3466    set base [[::constcl::car $data] numval]
3467    set end [expr {[[my length] numval] + $base - 1}]
3468    lrange $::constcl::vectorSpace $base $end
3469  }
```

The `value` method presents the store as a Tcl string.

(String instance) value (internal)	
Returns:	a Tcl string

```
3470  method value {} {
3471    # present the store as printable characters
3472    join [lmap c [my store] {$c char}] {}
3473  }
```

The `set!` method does nothing to a constant string. Given an index value which is between 0 and the length of the string, it changes the character at that position to a given character.

(String instance) set! (internal)	
k	a number
c	a character
Returns:	a string

```
3474  method set! {k c} {
3475    if {[my constant]} {
3476      ::error "string is constant"
3477    } else {
3478      set k [$k numval]
3479      if {$k < 0 ||
3480        $k >= [[my length] numval]} {
3481        ::error "index out of range\n$k"
3482      }
3483      set base [[::constcl::car $data] numval]
3484      lset ::constcl::vectorSpace $base+$k $c
3485    }
3486    return [self]
3487  }
```

The `fill!` method does nothing to a constant string. Given a character, it changes every character of the string to that character.

(String instance) fill! (internal)	
c	a character
Returns:	a string

```
3488  method fill! {c} {
3489    if {[my constant]} {
3490      ::error "string is constant"
3491    } else {
3492      set base [[::constcl::car $data] numval]
3493      set len [[my length] numval]
```

```
3494        for {set idx $base} \
3495          {$idx < $base+$len} \
3496          {incr idx} {
3497          lset ::constcl::vectorSpace $idx $c
3498        }
3499      }
3500      return [self]
3501    }
```

The substring method, given a *from* and a *to* index, returns the substring between those two indexes.

(String instance) substring (internal)	
from	a number
to	a number
Returns:	a Tcl string

```
3502    method substring {from to} {
3503      set f [$from numval]
3504      if {$f < 0 ||
3505        $f >= [[my length] numval]} {
3506        ::error "index out of range\n$f"
3507      }
3508      set t [$to numval]
3509      if {$t < 0 ||
3510        $t > [[my length] numval]} {
3511        ::error "index out of range\n$t"
3512      }
3513      if {$t < $f} {
3514        ::error "index out of range\n$t"
3515      }
3516      join [lmap c [
3517        lrange [my store] $f $t-1] {$c char}] {}
3518    }
```

The `mkconstant` method changes the instance from mutable to immutable.

(String instance) mkconstant (internal)	
Returns:	nothing

```
3519  method mkconstant {} {
3520    set constant 1
3521    return
3522  }
```

The `constant` method signals whether the string instance is mutable.

(String instance) constant (internal)	
Returns:	a Tcl truth value (1 or 0)

```
3523  method constant {} {
3524    return $constant
3525  }
```

The `external` method renders a string in external representation format.

(String instance) external (internal)	
Returns:	an external repr. of a string

```
3526  method external {} {
3527    return "\"[
3528      string map {\\ \\\\ \" \\\" \n \\n} [my value]]\""
3529  }
```

The `display` method prints a string in internal representation format.

(String instance) display (internal)	
port	an output port
Returns:	nothing

```
3530    method display {port} {
3531      $port put [my value]
3532    }
```

The `tstr` method yields the external representation of the string instance as a Tcl string. It is used by error messages and the `write` method.

(String instance) tstr (internal)	
Returns:	an external repr. of a string

```
3533    method tstr {} {
3534      return [my external]
3535    }
3536  }
```

MkString generator

`MkString` generates a String object.

MkString (internal)	
str	an external repr. of a string, w/o double quotes
Returns:	a string

```
3537  interp alias {} ::constcl::MkString \
3538    {} ::constcl::String new
```

string? procedure

`string?` recognizes a string by type.

string? (public)	
val	a value
Returns:	a boolean

```
reg string?

proc ::constcl::string? {val} {
  typeof? $val String
}
```

make-string procedure

`make-string` creates a string of *k* characters, optionally filled with *char* characters. If *char* is omitted, the string will be filled with space characters.

Example:

```
(let ((k 5))
  (make-string k))        ==>  "     "
(let ((k 5) (char #\A))
  (make-string k char))   ==>  "AAAAA"
```

make-string (public)	
k	a number
?char?	a character
Returns:	a string

```
reg make-string

proc ::constcl::make-string {k args} {
  set i [$k numval]
  if {[llength $args] == 0} {
    set char " "
  } else {
    lassign $args c
    set char [$c char]
  }
  return [MkString [::string repeat $char $i]]
}
```

string procedure

`string` constructs a string from a number of Lisp characters.
Example:

```
(string #\f #\o #\o)   ==>  "foo"
```

string (public)	
args	some characters
Returns:	a string

```
reg string

proc ::constcl::string {args} {
  set str {}
  foreach char $args {
    check {::constcl::char? $char} {
      CHAR expected\n([pn] [lmap c $args \
```

```
3563            {$c tstr}])
3564        }
3565        ::append str [$char char]
3566    }
3567    return [MkString $str]
3568 }
```

string-length procedure

`string-length` reports a string's length.

Example:

```
(string-length "foobar")   ==> 6
```

string-length (public)	
str	a string
Returns:	a number

```
3569 reg string-length
3570
3571 proc ::constcl::string-length {str} {
3572   check {::constcl::string? $str} {
3573     STRING expected\n([pn] [$str tstr])
3574   }
3575   return [$str length]
3576 }
```

string-ref procedure

`string-ref` yields the *k*-th character (0-based) in *str*.
Example:

```
(string-ref "foobar" 3)   ==> #\b
```

string-ref (public)	
str	a string
k	a number
Returns:	a character

```
reg string-ref

proc ::constcl::string-ref {str k} {
  check {::constcl::string? $str} {
    STRING expected\n([pn] [$str tstr] \
      [$k tstr])
  }
  check {::constcl::number? $k} {
    INTEGER expected\n([pn] [$str tstr] \
      [$k tstr])
  }
  return [$str ref $k]
}
```

string-set! procedure

`string-set!` replaces the character at *k* with *char* in a non-constant string.
Example:

```
(let ((str (string #\f #\o #\o))
      (k 2)
      (char #\x))
  (string-set! str k char))         ==>  "fox"
```

string-set! (public)	
str	a string
k	a number
char	a character
Returns:	a string

```
reg string-set!

proc ::constcl::string-set! {str k char} {
  check {string? $str} {
    STRING expected\n([pn] [$str tstr] [$k tstr] \
      [$char tstr])
  }
  check {number? $k} {
    INTEGER expected\n([pn] [$str tstr] \
      [$k tstr] [$char tstr])
  }
  check {char? $char} {
    CHAR expected\n([pn] [$str tstr] [$k tstr] \
      [$char tstr])
  }
  $str set! $k $char
  return $str
}
```

string=?, string-ci=?
string<?, string-ci<?

string>?, string-ci>?
string<=?, string-ci<=?
string>=?, string-ci>=?

The procedures `string=?`, `string<?`, `string>?`, `string<=?`, `string>=?` and their case insensitive variants `string-ci=?`, `string-ci<?`, `string-ci>?`, `string-ci<=?`, `string-ci>=?` compare strings.

string=?, string<?, string>? (public)	
str1	a string
str2	a string
Returns:	a boolean

string<=?, string>=? (public)	
str1	a string
str2	a string
Returns:	a boolean

string-ci=?, string-ci<?, string-ci>? (public)	
str1	a string
str2	a string
Returns:	a boolean

string-ci<=?, string-ci>=? (public)	
str1	a string
str2	a string
Returns:	a boolean

```
reg string=?

proc ::constcl::string=? {str1 str2} {
  check {string? $str1} {
    STRING expected\n([pn] [$str1 tstr] \
```

```
3613        [$str2 tstr])
3614    }
3615    check {string? $str2} {
3616      STRING expected\n([pn] [$str1 tstr] \
3617        [$str2 tstr])
3618    }
3619    if {[$str1 value] eq [$str2 value]} {
3620      return ${::#t}
3621    } else {
3622      return ${::#f}
3623    }
3624  }
```

```
3625  reg string-ci=?
3626
3627  proc ::constcl::string-ci=? {str1 str2} {
3628    check {string? $str1} {
3629      STRING expected\n([pn] [$str1 tstr] \
3630        [$str2 tstr])
3631    }
3632    check {string? $str2} {
3633      STRING expected\n([pn] [$str1 tstr] \
3634        [$str2 tstr])
3635    }
3636    if {[::string tolower [$str1 value]] eq
3637        [::string tolower [$str2 value]]} {
3638      return ${::#t}
3639    } else {
3640      return ${::#f}
3641    }
3642  }
```

```
reg string<?

proc ::constcl::string<? {str1 str2} {
  check {string? $str1} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
  check {string? $str2} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
  if {[$str1 value] < [$str2 value]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg string-ci<?

proc ::constcl::string-ci<? {str1 str2} {
  check {string? $str1} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
  check {string? $str2} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
  if {[::string tolower [$str1 value]] <
      [::string tolower [$str2 value]]} {
    return ${::#t}
  } else {
    return ${::#f}
```

```
  }
}
```

```
reg string>?

proc ::constcl::string>? {str1 str2} {
  check {string? $str1} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
  check {string? $str2} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
  if {[$str1 value] > [$str2 value]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

```
reg string-ci>?

proc ::constcl::string-ci>? {str1 str2} {
  check {string? $str1} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
  check {string? $str2} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
```

```
3706    if {[::string tolower [$str1 value]] >
3707        [::string tolower [$str2 value]]} {
3708      return ${::#t}
3709    } else {
3710      return ${::#f}
3711    }
3712  }
```

```
3713  reg string<=?
3714
3715  proc ::constcl::string<=? {str1 str2} {
3716    check {string? $str1} {
3717      STRING expected\n([pn] [$str1 tstr] \
3718        [$str2 tstr])
3719    }
3720    check {string? $str2} {
3721      STRING expected\n([pn] [$str1 tstr] \
3722        [$str2 tstr])
3723    }
3724    if {[$str1 value] <= [$str2 value]} {
3725      return ${::#t}
3726    } else {
3727      return ${::#f}
3728    }
3729  }
```

```
3730  reg string-ci<=?
3731
3732  proc ::constcl::string-ci<=? {str1 str2} {
3733    check {string? $str1} {
3734      STRING expected\n([pn] [$str1 tstr] \
3735        [$str2 tstr])
```

```
3736    }
3737    check {string? $str2} {
3738      STRING expected\n([pn] [$str1 tstr] \
3739        [$str2 tstr])
3740    }
3741    if {[::string tolower [$str1 value]] <=
3742        [::string tolower [$str2 value]]} {
3743      return ${::#t}
3744    } else {
3745      return ${::#f}
3746    }
3747  }
```

```
3748  reg string>=?
3749
3750  proc ::constcl::string>=? {str1 str2} {
3751    check {string? $str1} {
3752      STRING expected\n([pn] [$str1 tstr] \
3753        [$str2 tstr])
3754    }
3755    check {string? $str2} {
3756      STRING expected\n([pn] [$str1 tstr] \
3757        [$str2 tstr])
3758    }
3759    if {[$str1 value] >= [$str2 value]} {
3760      return ${::#t}
3761    } else {
3762      return ${::#f}
3763    }
3764  }
```

```
3765  reg string-ci>=?
```

```
3766
3767 proc ::constcl::string-ci>=? {str1 str2} {
3768   check {string? $str1} {
3769     STRING expected\n([pn] [$str1 tstr] \
3770       [$str2 tstr])
3771   }
3772   check {string? $str2} {
3773     STRING expected\n([pn] [$str1 tstr] \
3774       [$str2 tstr])
3775   }
3776   if {[::string tolower [$str1 value]] >=
3777       [::string tolower [$str2 value]]} {
3778     return ${::#t}
3779   } else {
3780     return ${::#f}
3781   }
3782 }
```

substring procedure

`substring` yields the substring of *str* that starts at *start* and ends before *end*.

Example:

```
(substring "foobar" 2 5)   ==> "oba"
```

substring (public)	
str	a string
start	a number
end	a number
Returns:	a string

```
reg substring

proc ::constcl::substring {str start end} {
  check {string? $str} {
    STRING expected\n([pn] [$str tstr] \
      [$start tstr] [$end tstr])
  }
  check {number? $start} {
    NUMBER expected\n([pn] [$str tstr] \
      [$start tstr] [$end tstr])
  }
  check {number? $end} {
    NUMBER expected\n([pn] [$str tstr] \
      [$start tstr] [$end tstr])
  }
  return [MkString [$str substring $start $end]]
}
```

string-append procedure

`string-append` joins strings together.

Example:

```
(string-append "foo" "bar")   ==>  "foobar"
```

string-append (public)	
args	some strings
Returns:	a string

```
reg string-append

```

```
proc ::constcl::string-append {args} {
    MkString [::append --> {*}[lmap arg $args {
      $arg value
    }]]
}
```

string->list procedure

`string->list` converts a string to a Lisp list of characters.
Example:

```
(string->list "foo")   ==>  (#\f #\o #\o)
```

string->list (public)	
str	a string
Returns:	a Lisp list of characters

```
reg string->list

proc ::constcl::string->list {str} {
  list {*}[$str store]
}
```

list->string procedure

`list->string` converts a Lisp list of characters to a string.
Example:

```
(list->string '(#\1 #\2 #\3))   ==> "123"
```

list->string (public)	
list	a Lisp list of characters
Returns:	a string

```
reg list->string

proc ::constcl::list->string {list} {
  MkString [::append --> {*}[
    lmap c [splitlist $list] {$c char}]]
}
```

string-copy procedure

`string-copy` makes a copy of a string.
Example:

```
(let ((str (string-copy "abc"))
      (k 0)
      (char #\x))
  (string-set! str k char))       ==>  "xbc"
```

string-copy (public)	
str	a string
Returns:	a string

```
reg string-copy

proc ::constcl::string-copy {str} {
  check {string? $str} {
    STRING expected\n([pn] [$str tstr])
```

```
3823    }
3824    return [MkString [$str value]]
3825  }
```

string-fill! procedure

`string-fill!` *str char* fills a non-constant string with *char*.
Example:

```
(let ((str (string-copy "foobar"))
      (char #\X))
  (string-fill! str char))           ==>  "XXXXXX"
```

string-fill! (public)	
str	a string
char	a character
Returns:	a string

```
3826  reg string-fill!
3827
3828  proc ::constcl::string-fill! {str char} {
3829    check {string? $str} {
3830      STRING expected\n([pn] [$str tstr] \
3831        [$char tstr])
3832    }
3833    $str fill! $char
3834    return $str
3835  }
```

8.9 Symbols

Symbols are like little immutable strings that are used to refer to things (variables, category labels, collection keys, etc) or for equality comparison against each other.

Symbol class

The Symbol class defines what capabilities a symbol has (in addition to those from the Base class), and also defines the internal representation of a symbol value expression. A symbol is stored in an instance as a Tcl string, and the `name` method yields the symbol's name as result.

```
3836 oo::class create ::constcl::Symbol {
3837   superclass ::constcl::Base
3838   variable name caseconstant
```

The Symbol constructor checks that the given name is a valid identifier and then stores it. It also sets `caseconstant` to 0, indicating that the name doesn't keep its case when turned into a string.

Symbol constructor (internal)	
n	a Tcl string
Returns:	nothing

```
3839   constructor {n} {
3840     ::constcl::idcheck $n
3841     set name $n
3842     set caseconstant 0
3843   }
```

The `name` method returns the symbol's name.

(Symbol instance) name (internal)	
Returns:	a Tcl string

```
3844  method name {} {
3845    set name
3846  }
```

The `value` method is a synonym for `name`.

(Symbol instance) value (internal)	
Returns:	a Tcl string

```
3847  method value {} {
3848    set name
3849  }
```

The = method compares the stored name with the name of a given symbol. It returns `#t` if they are equal, otherwise `#f`.

(Symbol instance) = (internal)	
sym	a symbol
Returns:	a boolean

```
3850  method = {symname} {
3851    if {$name eq [$sym name]} {
3852      return ${::#t}
3853    } else {
3854      return ${::#f}
3855    }
3856  }
```

The `constant` method signals whether the symbol instance is immutable (it is).

(Symbol instance) constant (internal)	
Returns:	a Tcl truth value (1)

```
3857  method constant {} {
3858    return 1
3859  }
```

The `case-constant` method signals whether the symbol instance is *case constant*, i.e. keeps its case when turned into a string.

(Symbol instance) case-constant (internal)	
Returns:	a Tcl truth value (1 or 0)

```
3860  method case-constant {} {
3861    set caseconstant
3862  }
```

The `make-case-constant` method makes the symbol *case constant*.

(Symbol instance) make-case-constant (internal)	
Returns:	a Tcl truth value (1)

```
3863  method make-case-constant {} {
3864    set caseconstant 1
3865  }
```

The `tstr` method yields the external representation of the symbol instance (the name) as a Tcl string. It is used by error messages.

(Symbol instance) tstr (internal)	
Returns:	a Tcl string

```
3866    method tstr {} {
3867      return $name
3868    }
3869  }
```

MkSymbol generator

`MkSymbol` generates a symbol with a given name. If a symbol with that name already exists, it is returned. Otherwise, a fresh symbol is created. Short form: `S`.

MkSymbol (internal)	
str	a Tcl string
Returns:	a symbol

```
3870  proc ::constcl::MkSymbol {str} {
3871    if {[dict exists $::constcl::symbolTable $str]} {
3872      return [dict get $::constcl::symbolTable $str]
3873    } else {
3874      set sym [::constcl::Symbol new $str]
3875      dict set ::constcl::symbolTable $str $sym
3876      return $sym
3877    }
3878  }
3879  interp alias {} S {} ::constcl::MkSymbol
```

symbol? procedure

`symbol?` recognizes a symbol by type.

symbol? (public)	
val	a value
Returns:	a boolean

```
reg symbol?

proc ::constcl::symbol? {val} {
  typeof? $val Symbol
}
```

symbol->string procedure

`symbol->string` yields a string consisting of the symbol name, usually lower-cased.

Example:

```
(let ((sym 'Foobar))
  (symbol->string sym))   ==>  "foobar"
```

symbol->string (public)	
sym	a symbol
Returns:	a string

```
reg symbol->string

proc ::constcl::symbol->string {sym} {
  check {symbol? $sym} {
```

```
3889      SYMBOL expected\n([pn] [$sym tstr])
3890    }
3891    if {![$sym case-constant]} {
3892      set str [MkString [
3893        ::string tolower [$sym name]]]
3894    } else {
3895      set str [MkString [$sym name]]
3896    }
3897    $str mkconstant
3898    return $str
3899  }
```

string->symbol procedure

`string->symbol` creates a symbol with the name given by the string. The symbol is 'case-constant', i.e. it will not be lower-cased.

Example:

```
(define sym (let ((str "Foobar"))
              (string->symbol str)))
sym                                    ==>  Foobar
(symbol->string sym)                   ==>  "Foobar"
```

string->symbol (public)	
str	a string
Returns:	a symbol

```
3900  reg string->symbol
3901
3902  proc ::constcl::string->symbol {str} {
```

```
3903    check {string? $str} {
3904      STRING expected\n([pn] [$obj tstr])
3905    }
3906    set sym [MkSymbol [$str value]]
3907    $sym make-case-constant
3908    return $sym
3909  }
```

8.10 Vectors

Vectors are heterogenous structures of fixed length whose elements are indexed by integers. The number of elements that a vector contains (the *length*) is set when the vector is created. Elements can be indexed by integers from zero to length minus one.

Vector class

The Vector class defines what capabilities a vector has (in addition to those from the Base class), and also defines the internal representation of a vector value expression. A vector is stored in an instance as a tuple of vector memory address and vector length. The `value` method yields the contents of the vector as result.

```
3910  oo::class create ::constcl::Vector {
3911    superclass ::constcl::Base
3912    variable data constant
```

The Vector constructor is divided in two main parts, one for the case where the value is a Lisp list, and one for the case where the value is a Tcl list. Their structure is similar: set the length (number of items), allocate vector space, and store the elements.

Vector constructor (internal)	
val	a value
Returns:	nothing

```
3913   constructor {val} {
3914     if {[T [::constcl::list? $val]]} {
3915       # if val is provided in the form of a Lisp list
3916       set len [[::constcl::length $val] numval]
3917       # allocate vector space for the elements
3918       set vsa [::constcl::vsAlloc $len]
3919       # store the elements in vector space
3920       set idx $vsa
3921       while {![T [::constcl::null? $val]]} {
3922         set elt [::constcl::car $val]
3923         lset ::constcl::vectorSpace $idx $elt
3924         incr idx
3925         set val [::constcl::cdr $val]
3926       }
3927     } else {
3928       # if val is provided in the form of a Tcl list
3929       set len [llength $val]
3930       # allocate vector space for the elements
3931       set vsa [::constcl::vsAlloc $len]
3932       # store the elements in vector space
3933       set idx $vsa
3934       foreach elt $val {
3935         lset ::constcl::vectorSpace $idx $elt
3936         incr idx
3937       }
```

```
3938      }
3939      # store the basic vector data: address of
3940      # first element and length
3941      set data [::constcl::cons [N $vsa] [N $len]]
3942      set constant 0
3943    }
```

The `baseadr` method returns the address of the first element as a number object.

(Vector instance) baseadr (internal)	
Returns:	a number

```
3944    method baseadr {} {
3945      ::constcl::car $data
3946    }
```

The `length` method returns the length (number of elements) as a number object.

(Vector instance) length (internal)	
Returns:	a number

```
3947    method length {} {
3948      ::constcl::cdr $data
3949    }
```

The `ref` method returns one element given the (0-based) index for it.

(Vector instance) ref (internal)	
k	a number
Returns:	a value

```
3950   method ref {k} {
3951     set k [$k numval]
3952     if {$k < 0 || $k >= [[my length] numval]} {
3953       ::error "index out of range\n$k"
3954     }
3955     lindex [my store] $k
3956   }
```

The `store` method returns the range of vector memory cells that store the vector's elements.

(Vector instance) store (internal)	
Returns:	a Tcl list of values

```
3957   method store {} {
3958     set base [[my baseadr] numval]
3959     set end [expr {[[my length] numval] + $base - 1}]
3960     lrange $::constcl::vectorSpace $base $end
3961   }
```

The `value` method is a synonym for `store`.

(Vector instance) value (internal)	
Returns:	a Tcl list of values

```
3962   method value {} {
3963     my store
3964   }
```

The `set!` method changes one element in a mutable vector given a (0-based) index value and a value.

(Vector instance) set! (internal)	
k	a number
val	a value
Returns:	a vector

```
3965  method set! {k val} {
3966    if {[my constant]} {
3967      ::error "vector is constant"
3968    } else {
3969      set k [$k numval]
3970      if {$k < 0 || $k >= [[my length] numval]} {
3971        ::error "index out of range\n$k"
3972      }
3973      set base [[my baseadr] numval]
3974      lset ::constcl::vectorSpace $k+$base $val
3975    }
3976    return [self]
3977  }
```

The `fill!` method changes every element in a mutable vector to a given value.

(Vector instance) fill! (internal)	
val	a value
Returns:	a vector

```
3978  method fill! {val} {
3979    if {[my constant]} {
3980      ::error "vector is constant"
3981    } else {
3982      set base [[my baseadr] numval]
3983      set len [[my length] numval]
3984      for {set idx $base} \
3985        {$idx < $len+$base} \
```

```
3986          {incr idx} {
3987          lset ::constcl::vectorSpace $idx $val
3988        }
3989      }
3990      return [self]
3991    }
```

The mkconstant method makes a vector immutable.

(Vector instance) mkconstant (internal)	
Returns:	a Tcl truth value (1)

```
3992    method mkconstant {} {
3993      set constant 1
3994    }
```

The constant method signals whether the vector instance is immutable.

(Vector instance) constant (internal)	
Returns:	a Tcl truth value (1 or 0)

```
3995    method constant {} {
3996      set constant
3997    }
```

The tstr method yields the external representation of the symbol instance (the name) as a Tcl string. It is used by error messages and for the write method.

(Vector instance) tstr (internal)	
Returns:	a Tcl string

```
3998    method tstr {} {
```

```
3999      return [format "#(%s)" [
4000        join [lmap val [my value] {$val tstr}]]]
4001    }
4002  }
```

MkVector generator

`MkVector` generates a Vector object.

MkVector (internal)	
vals	a Lisp or Tcl list of values
Returns:	a vector

```
4003  interp alias {} ::constcl::MkVector \
4004    {} ::constcl::Vector new
```

vector? procedure

`vector?` recognizes vectors by type.

vector? (public)	
val	a value
Returns:	a boolean

```
4005  reg vector?
4006
4007  proc ::constcl::vector? {val} {
4008    typeof? $val Vector
4009  }
```

make-vector procedure

`make-vector` creates a vector with a given length and optionally a fill value. If a fill value isn't given, the empty list will be used.

Example:

```
(let ((k 3))
  (make-vector k))        ==>  #(() () ())
(let ((k 3) (val #\A))
  (make-vector k val))    ==>  #(#\A #\A #\A)
```

make-vector? (public)	
k	a number
?val?	a value
Returns:	a vector

```
reg make-vector

proc ::constcl::make-vector {k args} {
  if {[llength $args] == 0} {
    set val ${::#NIL}
  } else {
    lassign $args val
  }
  MkVector [lrepeat [$k numval] $val]
}
```

vector procedure

Given a number of Lisp values, `vector` creates a vector containing them.

Example:

```
(vector 'a "foo" 99)   ==>  #(a "foo" 99)
```

vector (public)	
args	some values
Returns:	a vector

```
reg vector

proc ::constcl::vector {args} {
  MkVector $args
}
```

vector-length procedure

`vector-length` returns the length of a vector.

Example:

```
(vector-length '#(a "foo" 99))   ==>  3
```

vector-length (public)	
vec	a vector
Returns:	a number

```
reg vector-length

proc ::constcl::vector-length {vec} {
  check {vector? $vec} {
    VECTOR expected\n([pn] [$vec tstr])
  }
```

```
4031    return [$vec length]
4032  }
```

vector-ref procedure

`vector-ref` returns the element of *vec* at index *k* (0-based).
Example:

```
(let ((vec '#(a "foo" 99)) (k 1))
  (vector-ref vec k))              ==>  "foo"
```

vector-ref (public)	
vec	a vector
k	a number
Returns:	a value

```
4033  reg vector-ref
4034
4035  proc ::constcl::vector-ref {vec k} {
4036    check {vector? $vec} {
4037      VECTOR expected\n([pn] [$vec tstr] [$k tstr])
4038    }
4039    check {number? $k} {
4040      NUMBER expected\n([pn] [$vec tstr] [$k tstr])
4041    }
4042    return [$vec ref $k]
4043  }
```

vector-set! procedure

`vector-set!` sets the element at index *k* to *val* on a vector that isn't constant.

Example:

```
(let ((vec '#(a b c))
      (k 1)
      (val 'x))
  (vector-set! vec k val))      ==>  *error*
(let ((vec (vector 'a 'b 'c))
      (k 1)
      (val 'x))
  (vector-set! vec k val))      ==>  #(a x c)
```

vector-set! (public)	
vec	a vector
k	a number
val	a value
Returns:	a vector

```
reg vector-set!

proc ::constcl::vector-set! {vec k val} {
  check {vector? $vec} {
    VECTOR expected\n([pn] [$vec tstr] [$k tstr])
  }
  check {number? $k} {
    NUMBER expected\n([pn] [$vec tstr] [$k tstr])
  }
  return [$vec set! $k $val]
}
```

vector->list procedure

`vector->list` converts a vector value to a Lisp list.
Example:

```
(vector->list '#(a b c))   ==>  (a b c)
```

vector->list (public)	
vec	a vector
Returns:	a Lisp list of values

```
reg vector->list

proc ::constcl::vector->list {vec} {
  list {*}[$vec value]
}
```

list->vector procedure

`list->vector` converts a Lisp list value to a vector.
Example:

```
(list->vector '(1 2 3))   ==>  #(1 2 3)
```

list->vector (public)	
list	a Lisp list of values
Returns:	a vector

```
reg list->vector

proc ::constcl::list->vector {list} {
```

```
4063    vector {*}[splitlist $list]
4064  }
```

vector-fill! procedure

`vector-fill!` fills a non-constant vector with a given value.
Example:

```
(define vec (vector 'a 'b 'c))
(vector-fill! vec 'x)              ==>  #(x x x)
vec                                ==>  #(x x x)
```

vector-fill! (public)	
vec	a vector
fill	a value
Returns:	a vector

```
4065  reg vector-fill!
4066
4067  proc ::constcl::vector-fill! {vec fill} {
4068    check {vector? $vec} {
4069      VECTOR expected\n([pn] [$vec tstr] \
4070        [$fill tstr])
4071    }
4072    $vec fill! $fill
4073  }
```

9. Initialization

Before the interpreter can run, some elements must be initialized.

Vector space

Initialize the memory space for vector contents.

```
4074 set ::constcl::vectorSpaceSize [expr {1 * 1024}]
4075 set ::constcl::vectorSpace [
4076   lrepeat $::constcl::vectorSpaceSize [N 0]]
4077
4078 set ::constcl::vectorAssign 0
```

The `vsAlloc` procedure allocates vector space for strings and vectors. First it checks that there is enough space left, and then it increases the fill marker and returns its old value.

vsAlloc (internal)	
num	a number
Returns:	a Tcl number

```
proc ::constcl::vsAlloc {num} {
  if {$::constcl::vectorSpaceSize -
    $::constcl::vectorAssign < $num} {
    error "not enough vector space left"
  }
  set va $::constcl::vectorAssign
  incr ::constcl::vectorAssign $num
  return $va
}
```

Symbol table

Initialize the symbol table and gensym number.

```
unset -nocomplain ::constcl::symbolTable
set ::constcl::symbolTable [dict create]

set ::constcl::gensymnum 0
```

Recursion limit

Make it possible to reach (fact 100). Probably more than needed, but this amount can't hurt (default is 1000).

```
interp recursionlimit {} 2000
```

A set of source code constants

Pre-make a set of constants (e.g. `#NIL`, `#t`, and `#f`) and give them aliases for use in source text.

```
4093 set #NIL [::constcl::NIL new]
4094
4095 set #t [::constcl::True new]
4096
4097 set #f [::constcl::False new]
4098
4099 set #UNS [::constcl::Unspecified new]
4100
4101 set #UND [::constcl::Undefined new]
4102
4103 set #EOF [::constcl::EndOfFile new]
```

Pi and nil

Crown the definition register with the queen of numbers (or at least a double-precision floating point approximation).

```
4104 regvar pi [N 3.1415926535897931]
```

In this interpreter, `nil` does refer to the empty list.

```
4105 regvar nil ${::#NIL}
```

Environment startup

On startup, two `Environment` objects called `null_env` (the null environment, not the same as `null-environment` in Scheme) and `global_env` (the global environment) are created.

Make `null_env` empty and judgemental: this is where searches for unbound symbols end up.

```
::constcl::Environment create \
  ::constcl::null_env ${::#NIL} {}

oo::objdefine ::constcl::null_env {
  method find {sym} {
    self
  }
  method get {sym} {
    ::error "Unbound variable: [$sym name]"
  }
  method set {sym t_ i_} {
    ::error "Unbound variable: [$sym name]"
  }
}
```

Meanwhile, `global_env` is populated with all the definitions from the definitions register, `defreg`. This is where top level evaluation happens.

```
namespace eval ::constcl {
  Environment create global_env ${::#NIL} {} \
    ::constcl::null_env
  foreach v [dict values $defreg] {
    lassign $v key val
    lassign $val bt in
```

```
4126      global_env bind [S $key] $bt $in
4127    }
4128  }
```

Thereafter, each time a user-defined procedure is called, a new `Environment` object is created to hold the bindings introduced by the call, and also a link to the outer environment (the one closed over when the procedure was defined).

The Scheme base

Load the Scheme base to add more definitions to the global environment.

```
4129  pe {(load "schemebase.scm")}
```

10. A Scheme base

```
4130 ; An assortment of procedures to supplement the builtins.
```

get procedure

`get` is a procedure for picking out values out of property lists. It returns either the value or `#f` if the key isn't found.

get (public)	
plist	a Lisp list of values
key	a symbol
Returns:	a value OR #f

```
4131 (define (get plist key)
4132   (let ((v (memq key plist)))
4133     (if v
4134       (cadr v)
4135       #f)))
```

list-find-key procedure

`list-find-key` searches for a key in a property list. If it finds it, it returns the (0-based) index of it. If it doesn't find it, it returns -1. It doesn't look at the values.

list-find-key (public)	
lst	a Lisp list of values
key	a symbol
Returns:	a number

```
4136 (define (list-find-key lst key)
4137   (lfk lst key 0))
```

lfk procedure

`lfk` does the work for `list-find-key`.

lfk (public)	
lst	a Lisp list of values
key	a symbol
count	a number
Returns:	a number

```
4138 (define (lfk lst key count)
4139   (if (null? lst)
4140     -1
4141     (if (eq? (car lst) key)
4142       count
4143       (lfk (cddr lst) key (+ count 2)))))
```

list-set! procedure

`list-set!` works in analogy with `string-set!`. Given a list and an index, it finds the place to insert a value. Is in real trouble if the index value is out of range.

list-set! (public)	
lst	a Lisp list of values
idx	a number
val	a value
Returns:	a value

```
4144 (define (list-set! lst idx val)
4145   (if (zero? idx)
4146     (set-car! lst val)
4147     (list-set! (cdr lst) (- idx 1) val)))
```

delete! procedure

`delete!` removes a key-value pair from a property list. Returns the list.

delete! (public)	
lst	a Lisp list of values
key	a symbol
Returns:	a Lisp list of values

```
4148 (define (delete! lst key)
4149   (let ((idx (list-find-key lst key)))
4150     (if (< idx 0)
4151       lst
4152       (if (= idx 0)
```

```
4153          (set! lst (cddr lst))
4154          (let ((bef (del-seek lst (- idx 1)))
4155                (aft (del-seek lst (+ idx 2))))
4156            (set-cdr! bef aft))))
4157      lst))
```

del-seek procedure

`del-seek` does the searching for `delete!`.

del-seek (public)	
lst	a Lisp list of values
idx	a number
Returns:	a Lisp list of values

```
4158  (define (del-seek lst idx)
4159    (if (zero? idx)
4160      lst
4161      (del-seek (cdr lst) (- idx 1))))
```

get-alist procedure

`get-alist` is like `get` but for association lists.

get-alist (public)	
lst	a Lisp list of association pairs
key	a symbol
Returns:	a value

```
4162  (define (get-alist lst key)
4163    (let ((item (assq key lst)))
```

```
4164      (if item
4165        (cdr item)
4166        #f)))
```

pairlis procedure

`pairlis` takes two lists like `'(a b c)` and `'(1 2 3)` and produces a list of association pairs `'((a . 1) (b . 2) (c . 3))`.

pairlis (public)	
a	a Lisp list of values
b	a Lisp list of values
Returns:	a Lisp list of association pairs

```
4167 (define (pairlis a b)
4168   (if (null? a)
4169     '()
4170     (cons
4171       (cons (car a) (car b))
4172       (pairlis (cdr a) (cdr b)))))
```

set-alist! procedure

`set-alist!` updates a value in an association list, given a key.

set-alist! (public)	
lst	a Lisp list of association pairs
key	a symbol
val	a value
Returns:	a Lisp list of association pairs

```
4173 (define (set-alist! lst key val)
4174   (let ((item (assq key lst)))
4175     (if item
4176       (begin (set-cdr! item val) lst)
4177       lst)))
```

fact procedure

`fact` calculates the factorial of *n*. The function is obvious from the definition of factorial, but I've copied the code from Lispy.

fact (public)	
n	a number
Returns:	a number

```
4178 (define (fact n)
4179   (if (<= n 1)
4180     1
4181     (* n (fact (- n 1)))))
```

list-copy procedure

Returns a newly allocated copy of *list*. This copies each of the pairs comprising *list*. From MIT Scheme.

list-copy (public)	
list	a Lisp list of values
Returns:	a Lisp list of values

```
4182 (define (list-copy list)
```

```
(if (null? list)
  '()
  (cons (car list)
    (list-copy (cdr list)))))
```

And that's all. Thank you for joining me on this voyage of exploration.

Index