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:

---

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

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`

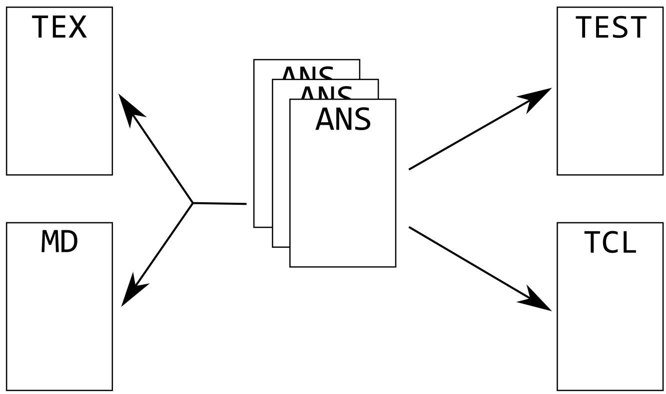
cheap computer, the following code (which calculates the factorial of 100) takes 0.04 seconds to run. That is thirteen times slower than Lispy assuming that Norvig's computer is as slow as mine, which is unlikely.

```
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:

```
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*.

```

proc regvar {name value} {
  if {![info exists ::constcl::defreg]} {
    set ::constcl::defreg [dict create]
  }
  set val [::list VARIABLE $value]
  set idx [dict size $::constcl::defreg]
  dict set ::constcl::defreg $idx [::list $name $val]
  return
}
```

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

```
reg atom?

proc ::constcl::atom? {val} {
  foreach type {symbol number string
      char boolean vector port eof} {
    if {[$type? $val] eq ${::#t}} {
```

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

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)

```
proc ::T {val} {
  if {$val eq ${::#f}} {
    return 0
  } else {
    return 1
  }
}
```

assert procedure

`assert` signals an error if an assertion fails.

assert (internal)	
expr	a Tcl expression
Returns:	nothing

```
proc assert {expr} {
  if {![uplevel [list expr $expr]]} {
    error "Failed assertion [
      uplevel [list subst $expr]]"
  }
}
```

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

```
proc ::constcl::pairlis-tcl {a b} {
  if {[T [null? $a]]} {
    parse {'()}
  } else {
    cons \
      [cons [car $a] [car $b]] \
      [pairlis-tcl [cdr $a] [cdr $b]]
  }
}
```

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

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

unbind procedure

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

unbind (internal)	
syms	some symbols
Returns:	nothing

```
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.

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

```
  } on error {} {
    set env ::constcl::global_env
  }
```

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.

```
  set syms $args
  foreach sym $syms {
    set e [$env find $sym]
    if {$e ne "::constcl::null_env"} {
      $e unbind $sym
    }
  }
}
```

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

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

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

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

```
proc ::constcl::splitlist {vals} {
  set result {}
  while {[T [pair? $vals]]} {
    lappend result [car $vals]
    set vals [cdr $vals]
  }
  return $result
}
```

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]
```

```
      set step [$t numval]
    }
  }
  set res $start
  while {$step > 0 && $end > [incr start $step] ||
      $step < 0 && $end < [incr start $step]} {
    lappend res $start
  }
  return [list {*}[lmap r $res {MkNumber $r}]]
}
```

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-

```
reg error

proc ::constcl::error {msg args} {
  set exprs $args
  if {[llength $exprs]} {
    set res [lmap expr $exprs {
      $expr tstr
    }]
    ::append msg " (" [join $res] ")"
  }
  ::error $msg
```

```
}
```

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

```
proc ::constcl::check {cond msg} {
  if {[uplevel $cond] eq ${::#f}} {
    ::error [
      uplevel [
        ::list subst [
          ::string trim $msg]]]
  }
}
```

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

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

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

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

```
  ::constcl::write [
    ::constcl::eval [
      ::constcl::read $port] $env]
}
```

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

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

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

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

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

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

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

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

parse procedure

`parse` only parses the input, returning an expression object.

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

```
proc ::parse {str} {
  ::constcl::read [
    ::constcl::MkStringInputPort $str]
}
```

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

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

w procedure

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

w (internal)	
val	a value
Returns:	nothing

```
proc ::w {val} {
  ::constcl::write $val
}
```

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

```
proc ::r {args} {
  ::constcl::read {*}$args
}
```

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

```
proc ::prw {str} {
  set expr [parse $str]
  set expr [::constcl::resolve-local-defines \
    [::constcl::cdr $expr]]
  ::constcl::write $expr
}
```

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

```
proc ::pxw {str {env ::constcl::global_env}} {
  set expr [parse $str]
  set op [::constcl::car $expr]
  lassign [::constcl::binding-info $op $env] btype hinfo
  if {$btype eq "SYNTAX"} {
    set expr [$hinfo $expr $env]
    ::constcl::write $expr
  } else {
```

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

```
    ::error "not a macro"
  }
}
```

1.3 Some small classes

Base class

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

```
catch { ::constcl::Base destroy }

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

```
  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

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

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

```
  method display {port} {
    my write $port
  }
```

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

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

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

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

```
  method tstr {} {
    return "#<base>"
  }
```

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

```
  method unknown {name args} {
    switch $name {
      car - cdr - set-car! -
      set-cdr {
        ::error "PAIR expected"
      }
      numval {
        ::error "NUMBER expected"
      }
    }
  }
}
```

Dot class

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

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

```
  method tstr {} {
    format "."
  }
}
```

dot? procedure

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

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

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

EndOfFile class

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

```
oo::singleton create ::constcl::EndOfFile {
  superclass ::constcl::Base
  method tstr {} {
    format "#<end-of-file>"
  }
}
```

eof? procedure

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

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

```
proc eof? {val} {
  if {$val eq ${::#EOF}} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

NIL class

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

```
oo::singleton create ::constcl::NIL {
  superclass ::constcl::Base
  method tstr {} {
    return "()"
  }
}
```

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.

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

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

```
reg read

proc ::constcl::read {args} {
  set c {}
  set unget {}
  set oldport $::constcl::Input_port
  if {[llength $args]} {
    lassign $args port
    set ::constcl::Input_port $port
  }
  set expr [read-expr]
  set ::constcl::Input_port $oldport
  return $expr
}
```

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

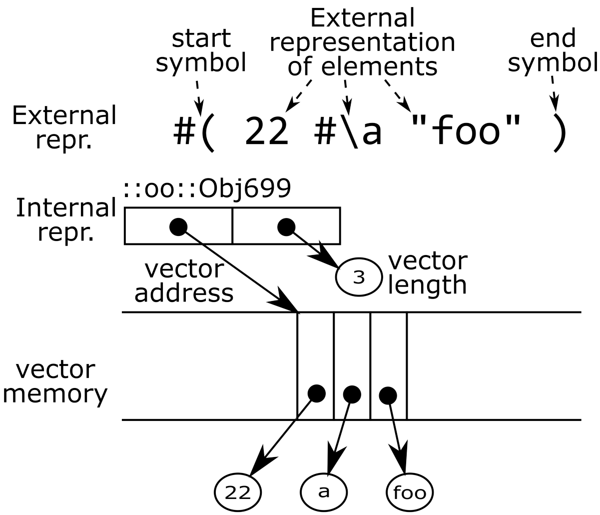
The external representation is a 'recipe' for an expression that expresses it in a unique and consistent way.
For example, the external representation for a vector is a pound sign (`#`), a left parenthesis (`(`), the external representation for some values, and a right parenthesis (`)`). When the reader/-parser 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:

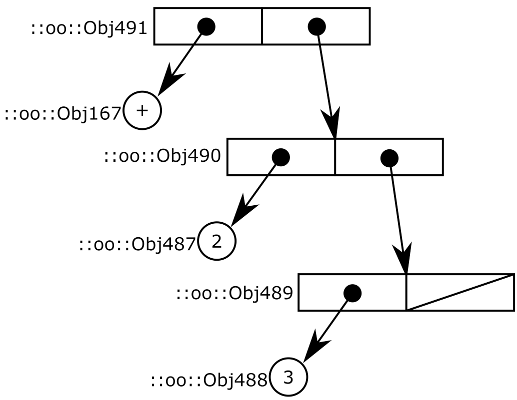
```
ConsTcl> (+ 2 3)
5
```

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 corresponding 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

```
proc ::constcl::read-expr {args} {
  upvar c c unget unget
  if {[llength $args]} {
    lassign $args c
  } else {
    set c [readchar]
  }
  set unget {}
  read-eof $c
  if {[::string is space $c] || $c eq ";"} {
    skip-ws
    read-eof $c
  }
  switch -regexp $c {
    {\"}          { read-string-expr }
    {\#}          { read-pound }
    {\'}          { read-quoted-expr }
    {\(}          { read-pair-expr ")" }
    {\+} - {\-}   { read-plus-minus $c }
    {\,}          { read-unquoted-expr }
    {\.} {
        set x [Dot new]; set c [readchar]; set x
    }
    {\[}          { read-pair-expr "\]" }
    {\`}          { read-quasiquoted-expr }
```

```
    {\d}          { read-number-expr $c }
    {^$}          { return #EOF }
    {[[:graph:]]} { read-identifier-expr $c }
    default {
      read-eof $c
      ::error "unexpected character ($c)"
    }
  }
}
```

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

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

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

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

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

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

```
    set c [readchar]
    while {[::string is alpha $c]} {
      ::append name $c
      set c [readchar]
    }
  }
```

Check if we have a valid character name.

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

Make and return a character object.

```
  set expr [MkChar $name]
  read-eof $expr
  return $expr
}
```

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

```
proc ::constcl::read-identifier-expr {args} {
  upvar c c unget unget
  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.

```
  if {[llength $args]} {
    set c [join $args {}]
  } else {
    set c [readchar]
  }
  read-eof $c
  set name {}
```

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

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

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

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

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

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

```
  # idcheck throws error if invalid identifier
  idcheck $name
  return [S $name]
}
```

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

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

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

```
  if {[llength $args]} {
    lassign $args c
  } else {
    set c [readchar]
  }
  read-eof $c
```

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

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

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

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

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

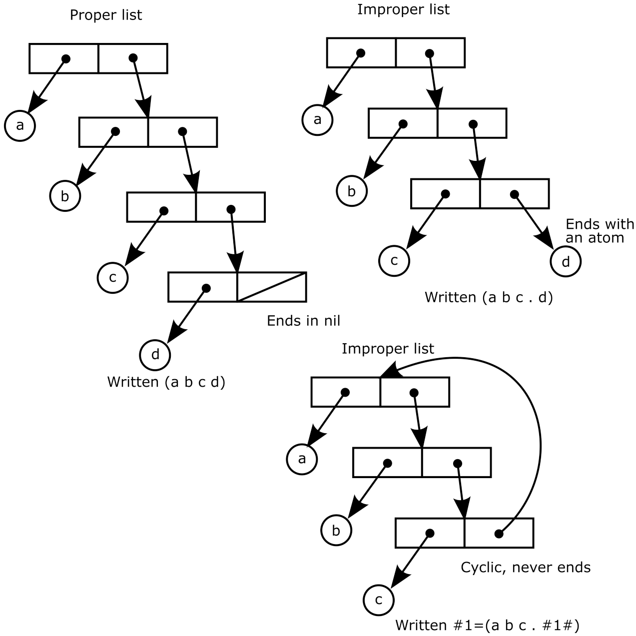
```
  check {::string is double -strict $num} {
      Invalid numeric constant $num
  }
  set expr [N $num]
  return $expr
}
```

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

```
proc ::constcl::read-pair-expr {char} {
  upvar c c unget unget
  set unget {}
  set expr [read-pair $char]
  read-eof $expr
  if {$c ne $char} {
```



A proper list and two improper ones.

```
    if {$char eq ")"} {
      ::error \
        "Missing right paren. ($c)."
    } else {
      ::error \
        "Missing right bracket ($c)."
```

```
    }
  } else {
    set unget {}
    set c [readchar]
  }
  return $expr
}
```

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

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

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

```
  set c [readchar]
```

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

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

```
  set a [read-expr $c]
  set res $a
  skip-ws
  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.

```
  while {![T [find-char? $char]]} {
    set x [read-expr $c]
    skip-ws
    read-eof $c
    if {[T [dot? $x]]} {
      set prev [read-expr $c]
      skip-ws
      read-eof $c
    } else {
      lappend res $x
    }
  }
```

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

```
  foreach r [lreverse $res] {
    set prev [cons $r $prev]
  }
  return $prev
}
```

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

```
proc ::constcl::read-plus-minus {char} {
  upvar c c unget unget
  set unget {}
  set c [readchar]
  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.

```
  if {[::string is digit -strict $c]} {
    set expr [read-number-expr $c]
    read-eof $expr
    if {$char eq "-"} {
      set expr [- $expr]
    }
    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.

```
  } elseif {[T [interspace? $c]] ||
      [T [delimiter? $c]]} {
    if {$char eq "+"} {
      return [S "+"]
    } else {
      return [S "-"]
    }
```

Otherwise, read an identifier.

```
  } else {
    set expr [read-identifier-expr $char $c]
    read-eof $expr
    return $expr
  }
}
```

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

```
proc ::constcl::read-pound {} {
  upvar c c unget unget
  set unget {}
  set c [readchar]
  read-eof $c
  switch $c {
    (    { set expr [read-vector-expr] }
    t    { if {[T [read-end?]]} {set expr ${::#t}} }
    f    { if {[T [read-end?]]} {set expr ${::#f}} }
    "\\" { set expr [read-character-expr] }
    default {
      ::error "Illegal #-literal: #$c"
    }
  }
  return $expr
}
```

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

```
proc ::constcl::read-quasiquoted-expr {} {
  upvar c c unget unget
  set unget {}
  set expr [read-expr]
  read-eof $expr
  make-constant $expr
  return [list [S quasiquote] $expr]
}
```

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

```
proc ::constcl::read-quoted-expr {} {
  upvar c c unget unget
  set unget {}
  set expr [read-expr]
  read-eof $expr
```

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

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

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

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

```
  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.

```
    if {$c eq "\\"} {
      ::append str $c
      set c [readchar]
      read-eof $c
    }
    ::append str $c
    set c [readchar]
  }
```

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

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

Create and return an immutable string object.

```
  set expr [MkString $str]
  make-constant $expr
  return $expr
}
```

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

```
proc ::constcl::read-unquoted-expr {} {
  upvar c c unget unget
  set unget {}
  set c [readchar]
  read-eof $c
  if {$c eq "@"} {
    set symbol "unquote-splicing"
    set expr [read-expr]
  } else {
    set symbol "unquote"
    set expr [read-expr $c]
  }
  read-eof $expr
  return [list [S $symbol] $expr]
}
```

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.

```
  set expr [MkVector $res]
  $expr mkconstant
  return $expr
}
```

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

```
proc ::constcl::make-constant {val} {
  if {[T [pair? $val]]} {
    $val mkconstant
    make-constant [car $val]
    make-constant [cdr $val]
  } elseif {[T [null? $val]]} {
    return
  } else {
    $val mkconstant
    return
  }
}
```

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

```
proc ::constcl::valid-char? {name} {
  if {[regexp {(?i)^#\\([[:graph:]]|space|newline)$} \
      $name]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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

```
proc ::constcl::readchar {} {
  upvar unget unget
  if {$unget ne {}} {
    set c $unget
    set unget {}
```

```
  } else {
    set c [$::constcl::Input_port get]
    if {[$::constcl::Input_port eof]} {
      return #EOF
    }
  }
  return $c
}
```

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

```
proc ::constcl::find-char? {char} {
  upvar c c unget unget
  # start with stored c
  while {[::string is space -strict $c]} {
    # this order seems strange but works
    set c [readchar]
    read-eof $c
    set unget $c
  }
  expr {($c eq $char) ? ${::#t} : ${::#f}}
}
```

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

```
proc ::constcl::read-end? {} {
  upvar c c unget unget
  set c [readchar]
  if {[T [interspace? $c]]} {
    return ${::#t}
  } elseif {[T [delimiter? $c]]} {
    set unget $c
    return ${::#t}
  } elseif {$c eq "#EOF"} {
    return #EOF
  } else {
    set unget $c
    return ${::#f}
  }
}
```

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

```
proc ::constcl::skip-ws {} {
  upvar c c unget unget
  while true {
    switch -regexp $c {
      {[[:space:]]} {
        set c [readchar]
      }
      {;} {
        while {$c ne "\n" && $c ne "#EOF"}  {
          set c [readchar]
        }
      }
      default {
        set unget $c
        return
      }
    }
  }
}
```

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

```
proc ::constcl::read-eof {args} {
  set chars $args
  foreach char $chars {
    if {$char eq "#EOF"} {
      return -level 1 -code return #EOF
    }
  }
}
```

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: `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

```
proc ::constcl::lookup {sym env} {
  lassign [binding-info $sym $env] type value
  if {$type eq "VARIABLE"} {
    return $value
  } else {
    error "not a variable name" $sym
  }
}
```

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

```
proc ::constcl::self-evaluating? {val} {
  if {[T [number? $val]] ||
    [T [string? $val]] ||
    [T [char? $val]] ||
    [T [boolean? $val]]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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

```
reg special quote

proc ::constcl::special-quote {expr env} {
  cadr $expr
}
```

3.4 Conditional

Example: `(if (> 99 100) (* 2 2) (+ 2 4))` ⇒ 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

```
reg special if

proc ::constcl::special-if {expr env} {
  set args [cdr $expr]
  if {[T [null? [cddr $args]]]} {
    if {[T [eval [car $args] $env]]} \
      {eval [cadr $args] $env}
  } else {
    if {[T [eval [car $args] $env]]} \
      {eval [cadr $args] $env} \
```

```
        {eval [caddr $args] $env}
    }
}
```

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

```
reg special case

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

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

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

```
    set datumlist [caar $clauses]
    set body [cdar $clauses]
    set predicate [list [S memv] $keyexpr \
        [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`.

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

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

```
    set env [MkEnv $env]
    /define [S predicate] $predicate $env
    /define [S body] $body $env
    /define [S rest] [
      do-case $keyexpr [cdr $clauses] $env] $env
    set qq "`(if ,predicate
               (begin ,@body)
               ,rest)"
    set expr [expand-quasiquote [parse $qq] $env]
    $env destroy
    return $expr
  }
}
```

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.

```
    set predicate [caar $clauses]
    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`.

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

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.

```
    if {[T [symbol? [car $body]]] &&
        [[car $body] name] eq "=>"} {
      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`.

```
    } else {
      if {[[length $body] numval] == 0} {
        set body $predicate
      } else {
        set body [cons [S begin] $body]
      }
    }
```

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

```
    set env [MkEnv $env]
    /define [S predicate] $predicate $env
    /define [S body] $body $env
    /define [S rest] [
      do-cond [cdr $clauses] $env] $env
    set qq "`(if ,predicate
               ,body
               ,rest)"
    set expr [expand-quasiquote [parse $qq] $env]
    $env destroy
    return $expr
  }
}
```

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

```
proc ::constcl::/begin {exps env} {
  if {[T [pair? $exps]]} {
    if {[T [pair? [cdr $exps]]]} {
      eval [car $exps] $env
      return [/begin [cdr $exps] $env]
    } else {
      return [eval [car $exps] $env]
    }
  } else {
    return [parse "'()"]
  }
}
```

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

```
proc ::constcl::rewrite-define {expr env} {
  if {[T [pair? [cadr $expr]]]} {
    set tail [cdr $expr]
    set env [::constcl::MkEnv $env]
    /define [S tail] $tail $env
    set qq "`(define ,(caar tail)
               (lambda ,(cdar tail) ,@(cdr tail)))"
    set expr [expand-quasiquote [parse $qq] $env]
    $env destroy
  }
  return $expr
}
```

/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

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

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

```
reg special set!

proc ::constcl::special-set! {expr env} {
  set args [cdr $expr]
  set var [car $args]
  set val [eval [cadr $args] $env]
  [$env find $var] assign $var VARIABLE $val
  set val
}
```

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)` ⇒ *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 an object 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

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

```
  }
}
```

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

```
reg special let

proc ::constcl::special-let {expr env} {
  if {[T [symbol? [cadr $expr]]]} {
```

```
    set expr [rewrite-named-let $expr $env]
  }
  set expr [rewrite-let $expr $env]
  eval $expr $env
}
```

rewrite-named-let procedure

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

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

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

```
  set variable [cadr $expr]
  set bindings [caddr $expr]
  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*.

```
  set vars [dict create $variable ${::#f}]
  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.

```
  set env [MkEnv $env]
  /define [S decl] [list {*}[dict values [
    dict map {k v} $vars {list $k $v}]]] $env
  /define [S variable] $variable $env
  /define [S varlist] [list {*}[lrange [
    dict keys $vars] 1 end]] $env
  /define [S body] $body $env
  /define [S call] [list {*}[
    dict keys $vars]] $env
  set qq "`(let ,decl
           (set!
             ,variable
               (lambda ,varlist ,@body)) ,call)"
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

rewrite-let procedure

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

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

The rewriter for regular `let` 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
```

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.

```
  set env [MkEnv $env]
  /define [S varlist] [list {*}[
    dict keys $vars]] $env
  /define [S body] $body $env
  /define [S vallist] [list {*}[
    dict values $vars]] $env
  set qq "`((lambda ,varlist ,@body)
            ,@vallist)"
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

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

```
proc ::constcl::parse-bindings {name bindings} {
  upvar $name vars
  foreach binding [splitlist $bindings] {
    set var [car $binding]
    set val [cadr $binding]
    if {$var in [dict keys $vars]} {
        ::error "'[$var name]' occurs more than once"
    }
    dict set vars $var $val
  }
  return
}
```

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.

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

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

```
set env [MkEnv $env]
# outer vars
/define [S ovars] [
  list {*}[dict keys $outer]] $env
# outer vals
/define [S ovals] [
  list {*}[dict values $outer]] $env
# inner vars
/define [S ivars] [
  list {*}[dict keys $inner]] $env
# inner vals
/define [S ivals] [
  list {*}[dict values $inner]] $env
/define [S assigns] [list {*}[lmap {k v} $assigns {
    list [S set!] $k $v
  }]] $env
/define [S body] $body $env
set qq "`((lambda ,ovars
           ((lambda ,ivars ,@assigns) ,@ivals)
           ,@body) ,@ovals)"
```

```
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

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

```
reg special let*

proc ::constcl::special-let* {expr env} {
  set expr [rewrite-let* [cadr $expr] [cddr $expr] $env]
  eval $expr $env
}
```

rewrite-let* procedure

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

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

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

```
    /define [S body] $body $env
    set qq "`(begin ,@body)"
    set expr [expand-quasiquote [parse $qq] $env]
  } 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.

```
    /define [S var] [caar $bindings] $env
    /define [S val] [cadar $bindings] $env
    /define [S rest] [rewrite-let* [cdr $bindings] \
      $body $env] $env
    set qq "`((lambda (,var)
                ,rest) ,val)"
    set expr [expand-quasiquote [parse $qq] $env]
  }
  $env destroy
```

```
    return $expr
}
```

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.

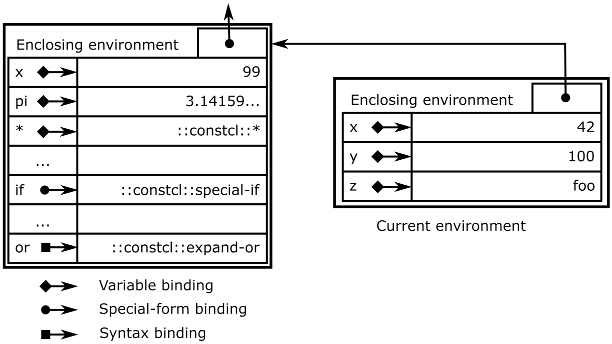
Environments make up the world the evaluator lives in and are the source of its values and procedures. The ability of procedure calls and execution of binding forms to temporarily change the current environment is a powerful one.

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

```
reg eval

proc ::constcl::eval \
  {expr {env ::constcl::global_env}} {
  if {[T [symbol? $expr]]} {
```

```
        lookup $expr $env
    } elseif {[T [self-evaluating? $expr]]} {
        set expr
    } elseif {[T [pair? $expr]]} {
        eval-form $expr $env
    } else {
        error "unknown expression type [$expr tstr]"
    }
}
```

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

```
proc ::constcl::eval-form {expr env} {
  set op [car $expr]
  set args [cdr $expr]
  if {[T [symbol? $op]]} {
    lassign [binding-info $op $env] btype hinfo
    switch $btype {
      UNBOUND {
        error "unbound symbol" $op
      }
      SPECIAL {
        $hinfo $expr $env
      }
      VARIABLE {
        invoke $hinfo [eval-list $args $env]
      }
      SYNTAX {
        eval [$hinfo $expr $env] $env
      }
      default {
        error "unrecognized binding type" $btype
      }
    }
  } else {
    invoke [eval $op $env] [eval-list $args $env]
  }
}
```

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

```
proc ::constcl::binding-info {op env} {
  set actual_env [$env find $op]
  if {$actual_env eq "::constcl::null_env"} {
    return [::list UNBOUND {}]
  } else {
    return [$actual_env get $op]
  }
}
```

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

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

```
  }
}
```

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

```
reg macro and

proc ::constcl::expand-and {expr env} {
  set tail [cdr $expr]
  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

```
reg macro del!

proc ::constcl::expand-del! {expr env} {
  set tail [cdr $expr]
  set env [MkEnv $env]
  if {[T [null? $tail]]} {
    ::error "too few arguments, 0 of 2"
  }
  /define [S listname] [car $tail] $env
  if {[T [null? [cdr $tail]]]} {
    ::error "too few arguments, 1 of 2"
  }
  /define [S key] [cadr $tail] $env
  set qq "`(set! ,listname
              (delete! ,listname ,key))"
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

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

```
reg macro for

proc ::constcl::expand-for {expr env} {
  set res [do-for [cdr $expr] $env]
  lappend res [parse "'()"]
  return [list [S begin] {*}$res]
}
```

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

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

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

```
  if {[T [number? $seq]]} {
    set seq [in-range $seq]
  } else {
    set seq [eval $seq $env]
  }
```

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

```
  if {[T [list? $seq]]} {
    set seq [splitlist $seq]
  } elseif {[T [string? $seq]]} {
    set seq [lmap c [split [$seq value] {}] {
      switch $c {
        " "  { MkChar #\\space }
        "\n" { MkChar #\\newline }
        default {
          MkChar #\\$c
        }
      }
    }]
  } elseif {[T [vector? $seq]]} {
    set seq [$seq value]
  } else {
    ::error "unknown sequence type [$seq tstr]"
  }
  return $seq
}
```

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

```
proc ::constcl::do-for {tail env} {
  # make clauses a Tcl list
  set clauses [splitlist [car $tail]]
  set body [cdr $tail]
  set data [dict create]
  set length 0
  foreach clause $clauses {
    set id [car $clause]
    set sequence [for-seq [cadr $clause] $env]
    set length [llength $sequence]
    # save every id and step of the iteration
    for {set i 0} {$i < $length} {incr i} {
        dict set data $id $i [lindex $sequence $i]
    }
  }
  set res {}
  # for every step of the iteration...
  for {set i 0} {$i < $length} {incr i} {
    set decl {}
```

```
    # retrieve the ids
    foreach id [dict keys $data] {
      # list the id and the step
      lappend decl [
        list $id [dict get $data $id $i]]
    }
    # add to the structure of let constructs
    lappend res [list [S let] [
        list {*}$decl] {*}[splitlist $body]]
  }
  return $res
}
```

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

```
reg macro for/and

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

```
}
```

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

```
reg macro for/list

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

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]
```

```
  } else {
    return [do-or $tail $env]
  }
}
```

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

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

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

```
reg macro quasiquote

proc ::constcl::expand-quasiquote {expr env} {
  set tail [cdr $expr]
  set qqlevel 0
  if {[T [list? [car $tail]]]} {
    set node [car $tail]
    return [qq-visit-child $node 0 $env]
  } elseif {[T [vector? [car $tail]]]} {
    set vect [car $tail]
    set res {}
    for {set i 0} {$i < [
        [vector-length $vect] numval]} {incr i} {
      set idx [MkNumber $i]
      set vecref [vector-ref $vect $idx]
      if {[T [pair? $vecref]] &&
          [T [eq? [car $vecref] [
            S unquote]]]} {
        if {$qqlevel == 0} {
          lappend res [eval [cadr $vecref] $env]
        }
```

```
      } 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]
```

```
        } else {
          lappend res [list [S unquote] [
            qq-visit-child [cadr $child] [
            expr {$qqlevel - 1}] $env]]
        }
      } elseif {[T [pair? $child]] &&
          [T [eq? [car $child] [
          S unquote-splicing]]]} {
        if {$qqlevel == 0} {
          lappend res {*}[splitlist [
            eval [cadr $child] $env]]
        }
      } elseif {[T [pair? $child]] &&
          [T [eq? [car $child] [S quasiquote]]]} {
        lappend res [list [S quasiquote] [car [
          qq-visit-child [cdr $child] [
            expr {$qqlevel + 1}] $env]]]
      } elseif {[T [atom? $child]]} {
        lappend res $child
      } else {
        lappend res [
          qq-visit-child $child $qqlevel $env]
      }
    }
  }
  return [list {*}$res]
}
```

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

```
reg macro unless

proc ::constcl::expand-unless {expr env} {
  set tail [cdr $expr]
  set env [MkEnv $env]
  /define [S tail] $tail $env
  set qq "`(if ,(car tail)
             '()
             (begin ,@(cdr tail)))"
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

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

```
reg macro when

proc ::constcl::expand-when {expr env} {
```

```
  set tail [cdr $expr]
  set env [MkEnv $env]
  /define [S tail] $tail $env
  set qq "`(if ,(car tail)
             (begin ,@(cdr tail))
             '())"
  set expr [expand-quasiquote [parse $qq] $env]
  $env destroy
  return $expr
}
```

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

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

```
  if {[T $error]} {
    return ${::#NIL}
  }
  set rest [lassign [
    extract-from-defines $exps VARS] v error]
  if {[T $error]} {
    return ${::#NIL}
  }
  if {$rest eq ${::#NIL}} {
    set rest [cons #UNS ${::#NIL}]
  }
  return [make-lambdas $v $a $rest]
}
```

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

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

```
        ![T [eq? [caar $exps] [S define]]]} {
      break
    }
    set n [car $exps]
    set k [length $n]
    if {![T [list? $n]] ||
        [$k numval] < 3 ||
        ![T [argument-list? [cadr $n]]] ||
        ([T [symbol? [cadr $n]]] &&
        [$k numval] > 3)} {
        return [::list ${::#NIL} ${::#t} ${::#NIL}]
      }
      if {[T [pair? [cadr $n]]]} {
        if {$part eq "VARS"} {
          set a [cons [caadr $n] $a]
        } else {
          set a [cons ${::#NIL} $a]
          set new [cons [cdadr $n] [cddr $n]]
          set new [cons [S lambda] $new]
          set-car! $a $new
        }
      } else {
        if {$part eq "VARS"} {
          set a [cons [cadr $n] $a]
        } else {
          set a [cons [caddr $n] $a]
        }
      }
      set exps [cdr $exps]
    }
    return [::list $a ${::#f} $exps]
}
```

argument-list? procedure

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

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

```
proc ::constcl::argument-list? {val} {
  if {$val eq ${::#NIL}} {
    return ${::#t}
  } elseif {[T [symbol? $val]]} {
    return ${::#t}
  } elseif {[T [atom? $val]]} {
    return ${::#f}
  }
  while {[T [pair? $val]]} {
    if {[symbol? [car $val]] eq ${::#f}} {
      return ${::#f}
    }
    set val [cdr $val]
  }
  if {$val eq ${::#NIL}} {
    return ${::#t}
  } elseif {[T [symbol? $val]]} {
    return ${::#t}
  }
}
```

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

```
proc ::constcl::make-lambdas {vars args body} {
  set tmps [make-temporaries $vars]
  set body [append-b [
    make-assignments $vars $tmps] $body]
  set body [cons $body ${::#NIL}]
  set n [cons $tmps $body]
  set n [cons [S lambda] $n]
  set n [cons $n $args]
  set n [cons $n ${::#NIL}]
  set n [cons $vars $n]
  set n [cons [S lambda] $n]
  set n [cons $n [make-undefineds $vars]]
  return $n
}
```

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

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

```
  while {$vals ne ${::#NIL}} {
    set res [cons [gensym "g"] $res]
    set vals [cdr $vals]
  }
  return $res
}
```

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

```
proc ::constcl::gensym {prefix} {
  set symbolnames [
    dict keys $::constcl::symbolTable]
  set s $prefix<[incr ::constcl::gensymnum]>
  while {$s in $symbolnames} {
    set s $prefix<[incr ::constcl::gensymnum]>
  }
  return [S $s]
}
```

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

```
proc ::constcl::append-b {a b} {
  if {$a eq ${::#NIL}} {
    return $b
  }
  set p $a
  while {$p ne ${::#NIL}} {
    if {[T [atom? $p]]} {
      ::error "append: improper list"
    }
    set last $p
    set p [cdr $p]
  }
  set-cdr! $last $b
  return $a
}
```

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

```
proc ::constcl::make-assignments {vars tmps} {
  set res ${::#NIL}
  while {$vars ne ${::#NIL}} {
    set asg [cons [car $tmps] ${::#NIL}]
    set asg [cons [car $vars] $asg]
    set asg [cons [S set!] $asg]
    set res [cons $asg $res]
    set vars [cdr $vars]
    set tmps [cdr $tmps]
  }
  return [cons [S begin] $res]
}
```

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

```
proc ::constcl::make-undefineds {vals} {
  set res ${::#NIL}
  while {$vals ne ${::#NIL}} {
    set res [cons [list [S quote] ${::#UND}] $res]
    set vals [cdr $vals]
  }
  return $res
}
```

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

```
reg write

proc ::constcl::write {val args} {
  if {$val ne ""} {
    set oldport $::constcl::Output_port
    if {[llength $args]} {
```

```
      lassign $args port
      set ::constcl::Output_port $port
    }
    $val write $::constcl::Output_port
    $::constcl::Output_port newline
    set ::constcl::Output_port $oldport
  }
  return
}
```

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

```
reg display

proc ::constcl::display {val args} {
  if {$val ne ""} {
    set oldport $::constcl::Output_port
    if {[llength $args]} {
      lassign $args port
      set ::constcl::Output_port $port
    }
    $val display $::constcl::Output_port
    $::constcl::Output_port flush
    set ::constcl::Output_port $oldport
  }
```

```
  return
}
```

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

```
proc ::constcl::write-pair {port pair} {
  # take an object and print the car
  # and the cdr of the stored value
  set a [car $pair]
  set d [cdr $pair]
  # print car
  $a write $port
  if {[T [pair? $d]]} {
    # cdr is a cons pair
    $port put " "
    write-pair $port $d
  } elseif {[T [null? $d]]} {
    # cdr is nil
    return
  } else {
    # it is an atom
    $port put " . "
    $d write $port
  }
  return
```

```
}
```

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.

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

```
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

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

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

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

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

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

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

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

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

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.

```
    } else {
      while true {
        if {[llength $vals] < 1} {
          error "too few arguments"
        }
        my bind [::constcl::car $syms] \
          [lindex $vals 0 0] [lindex $vals 0 1]
        set vals [lrange $vals 1 end]
        if {[T [
          ::constcl::symbol? [
            ::constcl::cdr $syms]]]} {
          my bind [::constcl::cdr $syms] \
```

```
          VARIABLE [
            ::constcl::list {*}[lmap v $vals {
              lindex $v 1
            }]]
        set vals {}
        break
      } else {
        set syms [::constcl::cdr $syms]
      }
    }
  }
```

Set the link to the outer environment.

```
    set outer_env $outer
  }
```

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

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

```
    } else {
      $outer_env find $sym
    }
  }
```

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

```
  method get {sym} {
    ::constcl::check {::constcl::symbol? $sym} {
      "SYMBOL expected\nEnvironment get"
    }
    dict get $bindings $sym
  }
```

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

```
  method unbind {sym} {
    ::constcl::check {::constcl::symbol? $sym} {
      "SYMBOL expected\nEnvironment unbind"
    }
    dict unset bindings $sym
    return
  }
```

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

```
method bind {sym type info} {
  ::constcl::check {::constcl::symbol? $sym} {
    "SYMBOL expected\nEnvironment bind"
  }
  if {[dict exists $bindings $sym]} {
    set bi [my get $sym]
    lassign $bi bt in
    if {$bt in {SPECIAL VARIABLE SYNTAX}} {
      error "[$sym name] is already bound"
    }
  }
  dict set bindings $sym [::list $type $info]
  return
}
```

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

```
  method assign {sym type info} {
    ::constcl::check {::constcl::symbol? $sym} {
      "SYMBOL expected\nEnvironment assign"
    }
    if {![dict exists $bindings $sym]} {
      error "[$sym name] is not bound"
    }
    set bi [my get $sym]
    lassign $bi bt in
    if {$bt ne "VARIABLE"} {
      error "[$sym name] is not assignable"
    }
    dict set bindings $sym [::list $type $info]
    return
  }
```

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

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

```
  method parent {} {
    set outer_env
  }
```

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

```
  method names {} {
    dict keys $bindings
  }
```

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

```
  method values {} {
    dict values $bindings
  }
```

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

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

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

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:

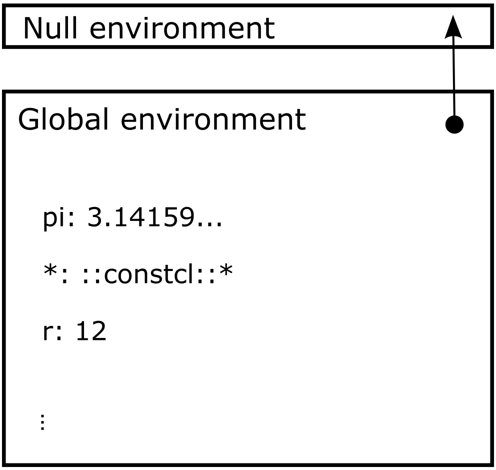
```
ConsTcl> (define (circle-area r) (* pi (* r r)))
ConsTcl> (circle-area 10)
314.1592653589793
```

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].

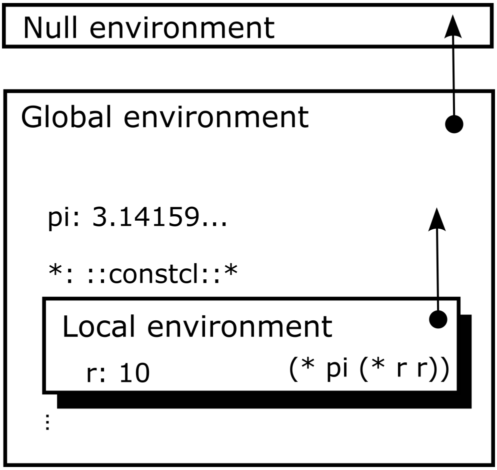
In the first image, we see the global environment before we call `circle-area` (and also the empty null environment which the global environment links to):

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

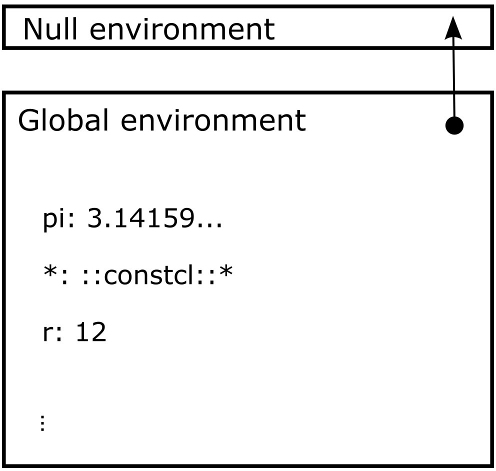
[15] See https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scope



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.

```
proc ::constcl::input {prompt} {
  puts -nonewline $prompt
  flush stdout
  set buf [gets stdin]
  set openpars [regexp -all -inline {\(} $buf]
  set clsepars [regexp -all -inline {\)} $buf]
  set openbrak [regexp -all -inline {\[} $buf]
  set clsebrak [regexp -all -inline {\]} $buf]
  while {[llength $openpars] > [llength $clsepars] ||
         [llength $openbrak] > [llength $clsebrak]} {
    ::append buf [gets stdin]
    set openpars [regexp -all -inline {\(} $buf]
```

```
    set clsepars [regexp -all -inline {\)} $buf]
    set openbrak [regexp -all -inline {\[} $buf]
    set clsebrak [regexp -all -inline {\]} $buf]
  }
  return $buf
}
```

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.

```
proc ::repl {{prompt "ConsTcl> "}} {
  set cur_env [::constcl::MkEnv ::constcl::global_env]
  set str [::constcl::input $prompt]
  while {$str ne ""} {
    set expr [parse $str]
    set val [::constcl::eval $expr $cur_env]
    ::constcl::write $val
    set str [::constcl::input $prompt]
  }
  $cur_env destroy
}
```

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

```
reg eq?

proc ::constcl::eq? {expr1 expr2} {
  if {[teq boolean? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } elseif {[teq symbol? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } elseif {[teq number? $expr1 $expr2] &&
      [veq $expr1 $expr2]} {
    return ${::#t}
  } elseif {[teq char? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } elseif {[teq null? $expr1 $expr2]} {
    return ${::#t}
  } elseif {[teq pair? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } elseif {[teq string? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } elseif {[teq vector? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } elseif {[teq procedure? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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)

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

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)

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

eqv? procedure

```
reg eqv?
```

```
proc ::constcl::eqv? {expr1 expr2} {
  if {[teq boolean? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } elseif {[teq symbol? $expr1 $expr2] &&
      [veq $expr1 $expr2]} {
    return ${::#t}
  } elseif {[teq number? $expr1 $expr2] &&
      [veq $expr1 $expr2]} {
    return ${::#t}
  } elseif {[teq char? $expr1 $expr2] &&
      [veq $expr1 eq $expr2]} {
    return ${::#t}
  } elseif {[teq null? $expr1 $expr2]} {
    return ${::#t}
  } elseif {[T [pair? $expr1]] &&
      [T [pair? $expr2]] &&
      [$expr1 car] eq [$expr2 car] &&
      [$expr1 cdr] eq [$expr2 cdr]} {
    return ${::#t}
  } elseif {[teq string? $expr1 $expr2] &&
      [veq $expr1 $expr2]} {
    return ${::#t}
  } elseif {[teq vector? $expr1 $expr2] &&
      [veq $expr1 $expr2]} {
    return ${::#t}
  } elseif {[teq procedure? $expr1 $expr2] &&
      $expr1 eq $expr2} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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.

```
oo::class create ::constcl::Number {
  superclass ::constcl::Base
  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

```
  constructor {val} {
    if {[::string is double -strict $val]} {
      set value $val
    } else {
      ::error "NUMBER expected\n$val"
    }
  }
```

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

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

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

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

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

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

```
  method positive? {} {
    if {$value > 0} {
      return ${::#t}
    } else {
      return ${::#f}
    }
  }
```

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

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

```
  method negative? {} {
    if {$value < 0} {
      return ${::#t}
    } else {
      return ${::#f}
    }
  }
```

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

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

```
method even? {} {
  if {$value % 2 == 0} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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

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

```
method odd? {} {
  if {$value % 2 == 1} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

The `value` method returns the stored number.

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

```
method value {} {
  set value
}
```

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

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

```
  method numval {} {
    set value
  }
```

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

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

```
  method constant {} {
    return 1
  }
```

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

```
  method tstr {} {
    return $value
  }
}
```

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

```
reg positive?

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

```
reg negative?

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

```
reg even?
```

```

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

```
reg odd?

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

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

```
reg abs

proc ::constcl::abs {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  if {[T [$num negative?]]} {
    return [N [expr {[$num numval] * -1}]]
  } else {
    return $num
  }
}
```

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} {
```

```
  set n [::tcl::mathop::% [[abs $num1] numval] \
    [[abs $num2] numval]]
  if {[T [$num1 negative?]]} {
    set n -$n
  }
  return [N $n]
}
```

modulo procedure

Example:

```
(modulo 7 3)   =>  1
```

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

```
reg modulo

proc ::constcl::modulo {num1 num2} {
  return [N [::tcl::mathop::% [$num1 numval] \
    [$num2 numval]]]
}
```

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} {
```

```
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::tan [$num numval]]
}
```

```
reg asin

proc ::constcl::asin {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::asin [$num numval]]
}
```

```
reg acos

proc ::constcl::acos {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::acos [$num numval]]
}
```

```
reg atan

proc ::constcl::atan {args} {
  if {[llength $args] == 1} {
    set num [lindex $args 0]
    check {number? $num} {
```

```
        NUMBER expected\n([pn] [$num tstr])
    }
    N [::tcl::mathfunc::atan [$num numval]]
  } else {
    lassign $args num1 num2
    check {number? $num1} {
        NUMBER expected\n([pn] [$num1 tstr])
    }
    check {number? $num2} {
        NUMBER expected\n([pn] [$num2 tstr])
    }
    N [::tcl::mathfunc::atan2 \
      [$num1 numval] [$num2 numval]]
  }
}
```

sqrt procedure

sqrt calculates the square root.

sqrt (public)	
num	a number
Returns:	a number

```
reg sqrt

proc ::constcl::sqrt {num} {
  check {number? $num} {
      NUMBER expected\n([pn] [$num tstr])
  }
  N [::tcl::mathfunc::sqrt [$num numval]]
}
```

expt procedure

expt calculates x to the power y, or x^y.

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

```
reg expt

proc ::constcl::expt {num1 num2} {
  check {number? $num1} {
      NUMBER expected\n([pn] [$num1 tstr] \
        [$num2 tstr])
  }
  check {number? $num2} {
      NUMBER expected\n([pn] [$num1 tstr] \
        [$num2 tstr])
  }
  N [::tcl::mathfunc::pow [$num1 numval] \
    [$num2 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`

```
proc frombase {base number} {
  set digits {0 1 2 3 4 5 6 7 8 9 A B C D E F}
  set negative [regexp ^-(.+) $number -> number]
  set res 0
  foreach digit [split $number {}] {
    # dv = decimal value
    set dv [lsearch $digits $digit]
    if {$dv < 0 || $dv >= $base} {
      ::error "bad digit $dv for base $base"
    }
    set res [expr {$res * $base + $dv}]
  }
  if $negative {set res -$res}
  set res
}
```

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).

```
oo::singleton create ::constcl::True {
  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

```
  method tstr {} {
    return "#t"
  }
}
```

```
oo::singleton create ::constcl::False {
  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

```
  method tstr {} {
    return "#f"
  }
}
```

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

```
proc ::constcl::MkBoolean {bool} {
  switch $bool {
    "#t" { return ${::#t} }
    "#f" { return ${::#f} }
    default { ::error "invalid boolean ($bool)" }
  }
}
```

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

```
reg boolean?

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

```
  }
}
```

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

```
reg not

proc ::constcl::not {val} {
  if {$val eq ${::#f}} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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.

```
oo::class create ::constcl::Char {
  superclass ::constcl::Base
  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

```
  constructor {val} {
    switch -regexp $val {
      {(?i)#\\space} {
        set val " "
      }
      {(?i)#\\newline} {
```

```
        set val "\n"
      }
      {#\\[[:graph:]]} {
        set val [::string index $val 2]
      }
      default {
        ::error "CHAR expected\n$val"
      }
    }
    set value $val
  }
```

The char method yields the stored character value.

(Char instance) char (internal)	
Returns:	a Tcl character

```
  method char {} {
    set value
  }
```

The alphabetic? method is a predicate which tests if the stored value is an alphabetic character.

(Char instance) alphabetic? (internal)	
Returns:	a boolean

```
  method alphabetic? {} {
    if {[::string is alpha $value]} {
      return ${::#t}
    } else {
      return ${::#f}
    }
  }
```

The `numeric?` method is a predicate which tests if the stored value is a numeric character.

(Char instance) numeric? (internal)	
Returns:	a boolean

```
method numeric? {} {
  if {[::string is digit $value]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

The `whitespace?` method is a predicate which tests if the stored value is a whitespace character.

(Char instance) whitespace? (internal)	
Returns:	a boolean

```
method whitespace? {} {
  if {[::string is space $value]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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

```
  method upper-case? {} {
    if {[::string is upper $value]} {
      return ${::#t}
    } else {
      return ${::#f}
    }
  }
```

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

```
  method lower-case? {} {
    if {[::string is lower $value]} {
      return ${::#t}
    } else {
      return ${::#f}
    }
  }
```

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

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

```
  method constant {} {
    return 1
  }
```

The `value` method is another way to yield the stored value

(Char instance) value (internal)	
Returns:	a Tcl character

```
  method value {} {
    return $value
  }
```

The `external` method translates the stored value back to external representation.

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

```
  method external {} {
    switch $value {
      " " {
        return "#\\space"
      }
      "\n" {
        return "#\\newline"
      }
      default {
        return "#\\$value"
      }
    }
  }
```

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

```
  method display {port} {
    $port put [my char]
  }
```

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

```
  method tstr {} {
    return [my external]
  }
}
```

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

```
proc ::constcl::MkChar {char} {
  if {[regexp -nocase {space|newline} $char]} {
      set char [::string tolower $char]
  }
  foreach instance [
    info class instances Char] {
    if {[$instance external] eq $char} {
      return $instance
    }
  }
  return [::constcl::Char new $char]
}
```

char? procedure

`char?` recognizes Char values by type.

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

```
reg char?

proc ::constcl::char? {val} {
  return [typeof? $val Char]
}
```

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]]} {
```

```
    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]]} {
    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]]} {
    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]]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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

```
reg char-alphabetic?

proc ::constcl::char-alphabetic? {char} {
  check {char? $char} {
    CHAR expected\n([pn] [$char tstr])
  }
  return [$char alphabetic?]
}
```

```
reg char-numeric?

proc ::constcl::char-numeric? {char} {
  check {char? $char} {
    CHAR expected\n([pn] [$char tstr])
  }
```

```
  return [$char numeric?]
}
```

```
reg char-whitespace?

proc ::constcl::char-whitespace? {char} {
  check {char? $char} {
    CHAR expected\n([pn] [$char tstr])
  }
  return [$char whitespace?]
}
```

```
reg char-upper-case?

proc ::constcl::char-upper-case? {char} {
  check {char? $char} {
    CHAR expected\n([pn] [$char tstr])
  }
  return [$char upper-case?]
}
```

```
reg char-lower-case?

proc ::constcl::char-lower-case? {char} {
  check {char? $char} {
    CHAR expected\n([pn] [$char tstr])
  }
  return [$char lower-case?]
}
```

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
```

```
  } else {
    return [MkChar [
      ::string toupper [$char external]]]
  }
}
```

```
reg char-downcase

proc ::constcl::char-downcase {char} {
  check {char? $char} {
    CHAR expected\n([pn] [$char tstr])
  }
  if {[$char char] in [::list " " "\n"]} {
    return $char
  } else {
    return [MkChar [
      ::string tolower [$char external]]]
  }
}
```

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.

```
catch { ::constcl::Procedure destroy }

oo::class create ::constcl::Procedure {
  superclass ::constcl::Base
  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

```
  constructor {p b e} {
    set parms $p
```

```
    set body $b
    set env $e
  }
}
```

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

```
oo::define ::constcl::Procedure method call {args} {
  set vals [lmap a $args {list VARIABLE $a}]
  ::constcl::eval $body [
    ::constcl::MkEnv $parms $vals $env]
}
```

The `value` method is a dummy.

(Procedure instance) value (internal)	
Returns:	nothing

```
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

```
oo::define ::constcl::Procedure method tstr {} {
  regexp {(\d+)} [self] -> num
  return "#<proc-$num>"
}
```

MkProcedure generator

`MkProcedure` generates a Procedure object.

MkProcedure (internal)	
parms	a Scheme formals list
body	an expression
env	an environment
Returns:	a procedure

```
interp alias {} ::constcl::MkProcedure \
  {} ::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

```
reg procedure?

proc ::constcl::procedure? {val} {
```

```
  if {[typeof? $val Procedure] eq ${::#t}} {
    return ${::#t}
  } elseif {[::string match "::constcl::*" $val]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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

```
reg apply

proc ::constcl::apply {pr vals} {
  check {procedure? $pr} {
    PROCEDURE expected\n([pn] [$pr tstr] ...)
  }
  invoke $pr $vals
}
```

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.

```
  set res {}
  for {set item 0} \
    {$item < [llength [lindex $arglists 0]]} \
    {incr item} {
    set actuals {}
    for {set arglist 0} \
      {$arglist < [llength $arglists]} \
      {incr arglist} {
      lappend actuals [
        lindex $arglists $arglist $item]
    }
    lappend res [invoke $pr [list {*}$actuals]]
  }
  return [list {*}$res]
}
```

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

```
reg for-each

proc ::constcl::for-each {proc args} {
  check {procedure? $proc} {
    PROCEDURE expected\n([pn] [$proc 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 `proc` is invoked on the Lisp list of actuals. After all the iterations, the empty list is returned.

```
  for {set item 0} \
    {$item < [llength [lindex $arglists 0]]} \
    {incr item} {
    set actuals {}
    for {set arglist 0} \
      {$arglist < [llength $arglists]} \
      {incr arglist} {
      lappend actuals [
        lindex $arglists $arglist $item]
    }
    invoke $proc [list {*}$actuals]
  }
  return ${::#NIL}
}
```

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

```
oo::abstract create ::constcl::Port {
  superclass ::constcl::Base
  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

```
  constructor {args} {
    if {[llength $args]} {
      lassign $args handle
    } else {
      set handle ${::#NIL}
    }
  }
```

The handle method yields the stored handle value.

(concrete instance) handle (internal)	
Returns:	a channel handle or NIL

```
  method handle {} {
    set handle
  }
```

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

```
  method close {} {
    close $handle
    set handle ${::#NIL}
    return
  }
}
```

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.

```
oo::class create ::constcl::InputPort {
  superclass ::constcl::Port
  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

```
  method open {name} {
    try {
      set handle [open [$name value] "r"]
    } on error {} {
      set handle ${::#NIL}
    }
    return $handle
  }
```

The get method reads one character from the channel of the stored handle.

(InputPort instance) get (internal)	
Returns:	a Tcl character

```
  method get {} {
    chan read $handle 1
  }
```

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)

```
  method eof {} {
    chan eof $handle
  }
```

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

```
  method copy {} {
    ::constcl::InputPort new $handle
  }
```

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

```
  method tstr {} {
    regexp {(\d+)} [self] -> num
    return "#<input-port-$num>"
  }
}
```

MkInputPort generator

`MkInputPort` generates an InputPort object.

MkInputPort (internal)	
?handle?	a channel handle
Returns:	an input port

```
interp alias {} ::constcl::MkInputPort \
  {} ::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.

```
oo::class create ::constcl::StringInputPort {
  superclass ::constcl::Port
  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

```
  constructor {str} {
    set buffer $str
    set read_eof 0
  }
```

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

```
  method open {name} {}
  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

```
method get {} {
  if {[::string length $buffer] == 0} {
    set read_eof 1
    return #EOF
  }
  set c [::string index $buffer 0]
  set buffer [::string range $buffer 1 end]
  return $c
}
```

The `eof` method reports end-of-file status on the buffer.

(StringInputPort instance) eof (internal)	
Returns:	a Tcl truth value (1 or 0)

```
method eof {} {
  return $read_eof
}
```

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

```
method copy {} {
  ::constcl::StringInputPort new $buffer
}
```

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

```
  method tstr {} {
    regexp {(\d+)} [self] -> num
    return "#<string-input-port-$num>"
  }
}
```

MkStringInputPort generator

`MkStringInputPort` generates a StringInputPort object.

MkStringInputPort (internal)	
str	a string
Returns:	a string input port

```
interp alias {} ::constcl::MkStringInputPort \
  {} ::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.

```
oo::class create ::constcl::OutputPort {
  superclass ::constcl::Port
  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

```
  method open {name} {
    ::error "remove this line to use"
    try {
      set handle [open [$name value] "w"]
    } on error {} {
      set handle ${::#NIL}
    }
    return $handle
  }
```

The OutputPort `put` method outputs a string on the channel in the stored handle.

(OutputPort instance) put (internal)	
str	a Tcl string
Returns:	nothing

```
method put {str} {
  puts -nonewline $handle $str
}
```

The `newline` method prints a newline on the channel in the stored handle.

(OutputPort instance) newline (internal)	
Returns:	nothing

```
method newline {} {
  puts $handle {}
}
```

The `flush` method flushes the output channel in the stored handle.

(OutputPort instance) flush (internal)	
Returns:	nothing

```
method flush {} {
  flush $handle
}
```

The copy method returns a new instance of OutputPort which is a copy of this instance, sharing the stored handle.

```
method copy {} {
  ::constcl::OutputPort new $handle
}
```

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

```
  method tstr {} {
    regexp {(\d+)} [self] -> num
    return "#<output-port-$num>"
  }
}
```

MkOutputPort generator

`MkOutputPort` generates an OutputPort object.

MkOutputPort (internal)	
?handle?	a channel handle
Returns:	an output port

```
interp alias {} ::constcl::MkOutputPort \
  {} ::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.

```
oo::class create ::constcl::StringOutputPort {
  superclass ::constcl::Port
  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

```
  constructor {args} {
    if {[llength $args]} {
      lassign $args str
      set buffer $str
    } else {
      set buffer {}
    }
  }
```

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

```
  method open {name} {}
  method close {} {}
```

The StringOutputPort `put` method appends a string to the internal buffer.

(StringOutputPort instance) put (internal)	
str	a Tcl string
Returns:	nothing

```
method put {str} {
  append buffer $str
  return
}
```

The `newline` method appends a newline character to the internal buffer.

(StringOutputPort instance) newline (internal)	
Returns:	nothing

```
method newline {} {
  append buffer \n
}
```

The `flush` method is present but does nothing.

(StringOutputPort instance) flush (internal)	
Returns:	nothing

```
method flush {} {}
```

The `tostring` method dumps the internal buffer as a string.

(StringOutputPort instance) tostring (internal)	
Returns:	a string

```
  method tostring {} {
    MkString $buffer
  }
```

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

```
  method copy {} {
    ::constcl::StringOutputPort new $buffer
  }
```

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

```
  method tstr {} {
    regexp {(\d+)} [self] -> num
    return "#<string-output-port-$num>"
  }
}
```

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

```
reg call-with-input-file

proc ::constcl::call-with-input-file {filename proc} {
  set port [open-input-file $filename]
  set res [invoke $proc [list $port]]
  close-input-port $port
  $port destroy
  return $res
}
```

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
```

```
    set ::constcl::Input_port $oldport
    close-input-port $newport
  }
  $newport destroy
}
```

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

```
reg with-output-to-file

proc ::constcl::with-output-to-file {filename thunk} {
  set newport [open-output-file $filename]
  if {[$newport handle] ne ${::#NIL}} {
    set oldport $::constcl::Output_port
    set ::constcl::Output_port $newport
    $thunk call
    set ::constcl::Output_port $oldport
    close-input-port $newport
  }
  $newport destroy
}
```

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
```

```

proc ::constcl::open-output-file {filename} {
  if {[file exists $filename]} {
    error "open-output-file: [fae] $filename"
  }
  set p [MkOutputPort]
  $p open $filename
  if {[$p handle] eq ${::#NIL}} {
    error "open-output-file: [cnof] $filename"
  }
  return $p
}
```

close-input-port procedure

`close-input-port` closes an input port.

close-input-port (public)	
port	an input port
Returns:	nothing

```
reg close-input-port

proc ::constcl::close-input-port {port} {
  if {[$port handle] eq "stdin"} {
    error "don't close the standard input port"
  }
  $port close
}
```

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]} {
```

```
    lassign $args port
  } else {
    set port [current-output-port]
  }
  $port newline
}
```

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

```
reg load

proc ::constcl::load {filename} {
  try {
    open-input-file $filename
  } on ok port {
  } on error {} {
    return
  }
  if {[$port handle] ne ${::#NIL}} {
    set expr [read $port]
    while {$expr ne "#EOF"} {
      eval $expr
      set expr [read $port]
    }
    close-input-port $port
  }
  $port destroy
```

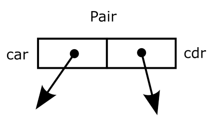
```
}
```

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:

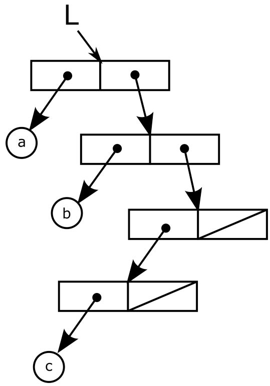
```
(cons 'a
  (cons 'b (cons (cons 'c '()) '())))  ==> (a b (c))
```

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.

```
oo::class create ::constcl::Pair {
  superclass ::constcl::Base
  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

```
constructor {a d} {
  set car $a
  set cdr $d
  set constant 0
}
```

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

(Pair instance) value (internal)	
Returns:	an external representation of a pair

```
method value {} {
  my tstr
}
```

The `car` method returns the value stored in the `car` variable.

(Pair instance) car (internal)	
Returns:	a value

```
method car {} {
  set car
}
```

The `cdr` method returns the value stored in the `cdr` variable.

(Pair instance) cdr (internal)	
Returns:	a value

```
method cdr {} {
  set cdr
}
```

The `set-car!` method modifies the value stored in the `car` variable.

(Pair instance) set-car! (internal)	
val	a value
Returns:	a pair

```
method set-car! {val} {
  ::constcl::check {my mutable?} {
    Can't modify a constant pair
  }
  set car $val
  self
}
```

The `set-cdr!` method modifies the value stored in the `cdr` variable.

(Pair instance) set-cdr! (internal)	
val	a value
Returns:	a pair

```
method set-cdr! {val} {
  ::constcl::check {my mutable?} {
    Can't modify a constant pair
  }
```

```
    set cdr $val
    self
  }
```

The `mkconstant` method changes the instance from mutable to immutable.

(Pair instance) mkconstant (internal)	
Returns:	nothing

```
  method mkconstant {} {
    set constant 1
    return
  }
```

The `constant` method signals whether the pair instance is immutable.

(Pair instance) constant (internal)	
Returns:	a Tcl truth value (1 or 0)

```
  method constant {} {
    return $constant
  }
```

The `mutable?` method is a predicate that tells if the pair instance is mutable or not.

(Pair instance) mutable? (internal)	
Returns:	a boolean

```
  method mutable? {} {
    expr {$constant ? ${::#f} : ${::#t}}
  }
```

The `write` method prints an external representation of the pair on the given port.

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

```
  method write {port} {
    $port put "("
    ::constcl::write-pair $port [self]
    $port put ")"
  }
```

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

```
  method tstr {} {
    format "(%s)" [::constcl::tstr-pair [self]]
  }
}
```

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]
```

```
  if {[T [pair? $d]]} {
    # cdr is a cons pair
    ::append str " "
    ::append str [tstr-pair $d]
  } elseif {[T [null? $d]]} {
    # cdr is nil
    return $str
  } else {
    # it is an atom
    ::append str " . "
    ::append str [$d tstr]
  }
  return $str
}
```

cons procedure

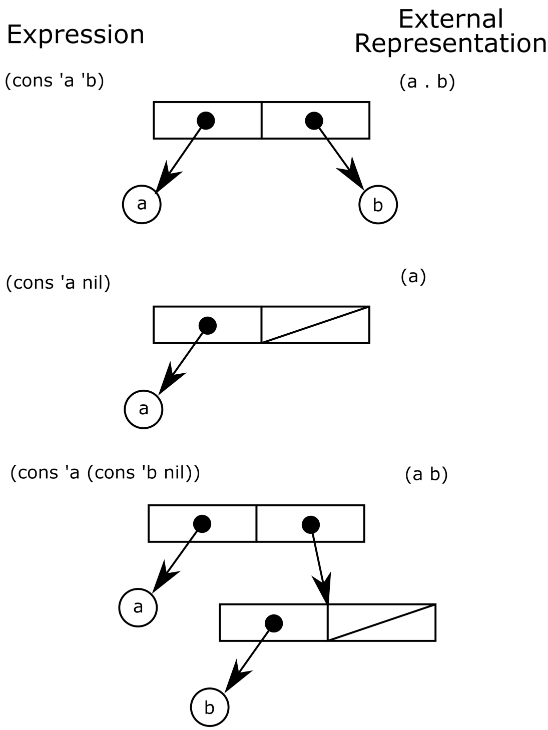
cons joins two values in a pair; useful in many operations such as pushing a new value onto a list.

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

```
reg cons
```



Examples of consing

```
proc ::constcl::cons {car cdr} {
```

```
  MkPair $car $cdr
}
```

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

```
reg car

proc ::constcl::car {pair} {
  $pair car
}
```

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} {
```

```
  upvar visited visited
  if {$pair in $visited} {
    return ${::#f}
  }
  lappend visited $pair
  if {[T [null? $pair]]} {
    return ${::#t}
  } elseif {[T [pair? $pair]]} {
    return [listp [cdr $pair]]
  } else {
    return ${::#f}
  }
}
```

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

```
reg list

proc ::constcl::list {args} {
  if {[llength $args] == 0} {
    return ${::#NIL}
  } else {
    set prev ${::#NIL}
```

```
    foreach obj [lreverse $args] {
      set prev [cons $obj $prev]
    }
    return $prev
  }
}
```

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

```
reg length

proc ::constcl::length {pair} {
  check {list? $pair} {
    LIST expected\n([pn] lst)
  }
  MkNumber [length-helper $pair]
}
```

length-helper procedure
`length-helper` is a helper procedure which measures a list recursively.

length-helper (internal)	
pair	a pair
Returns:	a Tcl number

```
proc ::constcl::length-helper {pair} {
  if {[T [null? $pair]]} {
    return 0
  } else {
    return [expr {1 +
      [length-helper [cdr $pair]]}]
  }
}
```

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

```
reg append

proc ::constcl::append {args} {
  set prev [lindex $args end]
  foreach r [lreverse [lrange $args 0 end-1]] {
    check {list? $r} {
      LIST expected\n([pn] [$r tstr])
```

```
    }
    set prev [copy-list $r $prev]
  }
  set prev
}
```

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

```
proc ::constcl::copy-list {pair next} {
  if {[T [null? $pair]]} {
    set next
  } elseif {[T [null? [cdr $pair]]]} {
    cons [car $pair] $next
  } else {
    cons [car $pair] [copy-list [cdr $pair] $next]
  }
}
```

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 {
```

```
      return [member-proc $epred $val1 [cdr $val2]]
    }
  }
}
```

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

```
reg assq

proc ::constcl::assq {val1 val2} {
  return [assoc-proc eq? $val1 $val2]
}
```

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

```
proc ::constcl::assoc-proc {epred val1 val2} {
  switch $epred {
    eq? { set name "assq" }
    eqv? { set name "assv" }
    equal? { set name "assoc" }
  }
  check {list? $val2} {
    LIST expected\n($name [$val1 tstr] [$val2 tstr])
  }
  if {[T [null? $val2]]} {
    return ${::#f}
  } elseif {[T [pair? $val2]]} {
    if {[T [pair? [car $val2]]] &&
      [T [$epred $val1 [caar $val2]]]} {
      return [car $val2]
    } else {
      return [assoc-proc $epred $val1 [cdr $val2]]
    }
  }
}
```

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`.

```
oo::class create ::constcl::String {
  superclass ::constcl::Base
  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

```
  constructor {val} {
```

```
    set val [string map {\\\\ \\ \\\" \" \\n \n} $val]
    set len [::string length $val]
    # allocate vector space for the string's
    # characters
    set vsa [::constcl::vsAlloc $len]
    # store the characters in vector space, as
    # Char objects
    set idx $vsa
    foreach elt [split $val {}] {
      if {$elt eq " "} {
        set c #\\space
      } elseif {$elt eq "\n"} {
        set c #\\newline
      } else {
        set c #\\$elt
      }
      lset ::constcl::vectorSpace $idx \
        [::constcl::MkChar $c]
      incr idx
    }
    # store the basic vector data: address of
    # first character and length
    set data [
      ::constcl::cons [N $vsa] [N $len]]
    set constant 0
  }
```

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)

```
  method = {str} {
```

```
    ::string equal [my value] [$str value]
  }
```

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

```
  method cmp {str} {
    ::string compare [my value] [$str value]
  }
```

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

```
  method length {} {
    ::constcl::cdr $data
  }
```

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

```
  method ref {k} {
    set k [$k numval]
    if {$k < 0 || $k >= [[my length] numval]} {
      ::error "index out of range\n$k"
    }
    lindex [my store] $k
  }
```

The `store` method presents the range in vector memory where the string is stored.

(String instance) store (internal)	
Returns:	a Tcl list of characters

```
  method store {} {
    set base [[::constcl::car $data] numval]
    set end [expr {[[my length] numval] + $base - 1}]
    lrange $::constcl::vectorSpace $base $end
  }
```

The `value` method presents the store as a Tcl string.

(String instance) value (internal)	
Returns:	a Tcl string

```
  method value {} {
    # present the store as printable characters
    join [lmap c [my store] {$c char}] {}
  }
```

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

```
method set! {k c} {
  if {[my constant]} {
    ::error "string is constant"
  } else {
    set k [$k numval]
    if {$k < 0 ||
      $k >= [[my length] numval]} {
      ::error "index out of range\n$k"
    }
    set base [[::constcl::car $data] numval]
    lset ::constcl::vectorSpace $base+$k $c
  }
  return [self]
}
```

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

```
method fill! {c} {
  if {[my constant]} {
    ::error "string is constant"
  } else {
    set base [[::constcl::car $data] numval]
    set len [[my length] numval]
```

```
      for {set idx $base} \
        {$idx < $base+$len} \
        {incr idx} {
        lset ::constcl::vectorSpace $idx $c
      }
    }
    return [self]
  }
```

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

```
  method substring {from to} {
    set f [$from numval]
    if {$f < 0 ||
      $f >= [[my length] numval]} {
      ::error "index out of range\n$f"
    }
    set t [$to numval]
    if {$t < 0 ||
      $t > [[my length] numval]} {
      ::error "index out of range\n$t"
    }
    if {$t < $f} {
      ::error "index out of range\n$t"
    }
    join [lmap c [
      lrange [my store] $f $t-1] {$c char}] {}
  }
```

The `mkconstant` method changes the instance from mutable to immutable.

(String instance) mkconstant (internal)	
Returns:	nothing

```
method mkconstant {} {
  set constant 1
  return
}
```

The `constant` method signals whether the string instance is mutable.

(String instance) constant (internal)	
Returns:	a Tcl truth value (1 or 0)

```
method constant {} {
  return $constant
}
```

The `external` method renders a string in external representation format.

(String instance) external (internal)	
Returns:	an external repr. of a string

```
method external {} {
  return "\"[
    string map {\\ \\\\ \" \\\" \n \\n} [my value]]\""
}
```

The `display` method prints a string in internal representation format.

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

```
  method display {port} {
    $port put [my value]
  }
```

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

```
  method tstr {} {
    return [my external]
  }
}
```

MkString generator

`MkString` generates a String object.

MkString (internal)	
str	an external repr. of a string, w/o double quotes
Returns:	a string

```
interp alias {} ::constcl::MkString \
  {} ::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 \
```

```
        {$c tstr}])
    }
    ::append str [$char char]
  }
  return [MkString $str]
}
```

string-length procedure

`string-length` reports a string's length.

Example:

```
(string-length "foobar")   ==> 6
```

string-length (public)	
str	a string
Returns:	a number

```
reg string-length

proc ::constcl::string-length {str} {
  check {::constcl::string? $str} {
    STRING expected\n([pn] [$str tstr])
  }
  return [$str length]
}
```

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] \
```

```
      [$str2 tstr])
  }
  check {string? $str2} {
    STRING expected\n([pn] [$str1 tstr] \
      [$str2 tstr])
  }
  if {[$str1 value] eq [$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]] eq
      [::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] \
      [$str2 tstr])
  }
  if {[::string tolower [$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])
  }
```

```
  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] \
      [$str2 tstr])
  }
  if {[::string tolower [$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] \
      [$str2 tstr])
  }
  if {[::string tolower [$str1 value]] >=
      [::string tolower [$str2 value]]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

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])
```

```
  }
  return [MkString [$str value]]
}
```

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

```
reg string-fill!

proc ::constcl::string-fill! {str char} {
  check {string? $str} {
    STRING expected\n([pn] [$str tstr] \
      [$char tstr])
  }
  $str fill! $char
  return $str
}
```

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.

```
oo::class create ::constcl::Symbol {
  superclass ::constcl::Base
  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

```
  constructor {n} {
    ::constcl::idcheck $n
    set name $n
    set caseconstant 0
  }
```

The `name` method returns the symbol's name.

(Symbol instance) name (internal)	
Returns:	a Tcl string

```
method name {} {
  set name
}
```

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

(Symbol instance) value (internal)	
Returns:	a Tcl string

```
method value {} {
  set name
}
```

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

```
method = {symname} {
  if {$name eq [$sym name]} {
    return ${::#t}
  } else {
    return ${::#f}
  }
}
```

The `constant` method signals whether the symbol instance is immutable (it is).

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

```
method constant {} {
  return 1
}
```

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)

```
method case-constant {} {
  set caseconstant
}
```

The `make-case-constant` method makes the symbol *case constant*.

(Symbol instance) make-case-constant (internal)	
Returns:	a Tcl truth value (1)

```
method make-case-constant {} {
  set caseconstant 1
}
```

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

```
  method tstr {} {
    return $name
  }
}
```

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

```
proc ::constcl::MkSymbol {str} {
  if {[dict exists $::constcl::symbolTable $str]} {
    return [dict get $::constcl::symbolTable $str]
  } else {
    set sym [::constcl::Symbol new $str]
    dict set ::constcl::symbolTable $str $sym
    return $sym
  }
}
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} {
```

```
    SYMBOL expected\n([pn] [$sym tstr])
  }
  if {![$sym case-constant]} {
    set str [MkString [
      ::string tolower [$sym name]]]
  } else {
    set str [MkString [$sym name]]
  }
  $str mkconstant
  return $str
}
```

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 lowercased.

Example:

```
(define sym (let ((str "Foobar"))
              (string->symbol str)))
sym                                      ==>  Foobar
(symbol->string sym)                     ==>  "Foobar"
```

string->symbol (public)	
str	a string
Returns:	a symbol

```
reg string->symbol

proc ::constcl::string->symbol {str} {
```

```
  check {string? $str} {
    STRING expected\n([pn] [$obj tstr])
  }
  set sym [MkSymbol [$str value]]
  $sym make-case-constant
  return $sym
}
```

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.

```
oo::class create ::constcl::Vector {
  superclass ::constcl::Base
  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

```
constructor {val} {
  if {[T [::constcl::list? $val]]} {
    # if val is provided in the form of a Lisp list
    set len [[::constcl::length $val] numval]
    # allocate vector space for the elements
    set vsa [::constcl::vsAlloc $len]
    # store the elements in vector space
    set idx $vsa
    while {![T [::constcl::null? $val]]} {
      set elt [::constcl::car $val]
      lset ::constcl::vectorSpace $idx $elt
      incr idx
      set val [::constcl::cdr $val]
    }
  } else {
    # if val is provided in the form of a Tcl list
    set len [llength $val]
    # allocate vector space for the elements
    set vsa [::constcl::vsAlloc $len]
    # store the elements in vector space
    set idx $vsa
    foreach elt $val {
      lset ::constcl::vectorSpace $idx $elt
      incr idx
    }
```

```
    }
    # store the basic vector data: address of
    # first element and length
    set data [::constcl::cons [N $vsa] [N $len]]
    set constant 0
  }
```

The baseadr method returns the address of the first element as a number object.

(Vector instance) baseadr (internal)	
Returns:	a number

```
  method baseadr {} {
    ::constcl::car $data
  }
```

The length method returns the length (number of elements) as a number object.

(Vector instance) length (internal)	
Returns:	a number

```
  method length {} {
    ::constcl::cdr $data
  }
```

The ref method returns one element given the (0-based) index for it.

(Vector instance) ref (internal)	
k	a number
Returns:	a value

```
method ref {k} {
  set k [$k numval]
  if {$k < 0 || $k >= [[my length] numval]} {
    ::error "index out of range\n$k"
  }
  lindex [my store] $k
}
```

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

```
method store {} {
  set base [[my baseadr] numval]
  set end [expr {[[my length] numval] + $base - 1}]
  lrange $::constcl::vectorSpace $base $end
}
```

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

(Vector instance) value (internal)	
Returns:	a Tcl list of values

```
method value {} {
  my store
}
```

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

```
method set! {k val} {
  if {[my constant]} {
    ::error "vector is constant"
  } else {
    set k [$k numval]
    if {$k < 0 || $k >= [[my length] numval]} {
      ::error "index out of range\n$k"
    }
    set base [[my baseadr] numval]
    lset ::constcl::vectorSpace $k+$base $val
  }
  return [self]
}
```

The `fill!` method changes every element in a mutable vector to a given value.

(Vector instance) fill! (internal)	
val	a value
Returns:	a vector

```
method fill! {val} {
  if {[my constant]} {
    ::error "vector is constant"
  } else {
    set base [[my baseadr] numval]
    set len [[my length] numval]
    for {set idx $base} \
      {$idx < $len+$base} \
```

```
        {incr idx} {
        lset ::constcl::vectorSpace $idx $val
      }
    }
    return [self]
  }
```

The `mkconstant` method makes a vector immutable.

(Vector instance) mkconstant (internal)	
Returns:	a Tcl truth value (1)

```
  method mkconstant {} {
    set constant 1
  }
```

The `constant` method signals whether the vector instance is immutable.

(Vector instance) constant (internal)	
Returns:	a Tcl truth value (1 or 0)

```
  method constant {} {
    set constant
  }
```

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

```
  method tstr {} {
```

```
    return [format "#(%s)" [
      join [lmap val [my value] {$val tstr}]]]
  }
}
```

MkVector generator

`MkVector` generates a Vector object.

MkVector (internal)	
vals	a Lisp or Tcl list of values
Returns:	a vector

```
interp alias {} ::constcl::MkVector \
  {} ::constcl::Vector new
```

vector? procedure

`vector?` recognizes vectors by type.

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

```
reg vector?

proc ::constcl::vector? {val} {
  typeof? $val Vector
}
```

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])
  }
```

```
  return [$vec length]
}
```

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

```
reg vector-ref

proc ::constcl::vector-ref {vec k} {
  check {vector? $vec} {
    VECTOR expected\n([pn] [$vec tstr] [$k tstr])
  }
  check {number? $k} {
    NUMBER expected\n([pn] [$vec tstr] [$k tstr])
  }
  return [$vec ref $k]
}
```

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} {
```

```
  vector {*}[splitlist $list]
}
```

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

```
reg vector-fill!

proc ::constcl::vector-fill! {vec fill} {
  check {vector? $vec} {
    VECTOR expected\n([pn] [$vec tstr] \
      [$fill tstr])
  }
  $vec fill! $fill
}
```

9. Initialization

Before the interpreter can run, some elements must be initialized.

Vector space

Initialize the memory space for vector contents.

```
set ::constcl::vectorSpaceSize [expr {1 * 1024}]
set ::constcl::vectorSpace [
  lrepeat $::constcl::vectorSpaceSize [N 0]]

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.

```
set #NIL [::constcl::NIL new]

set #t [::constcl::True new]

set #f [::constcl::False new]

set #UNS [::constcl::Unspecified new]

set #UND [::constcl::Undefined new]

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).

```
regvar pi [N 3.1415926535897931]
```

In this interpreter, `nil` does refer to the empty list.

```
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
```

```
    global_env bind [S $key] $bt $in
  }
}
```

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.

```
pe {(load "schemebase.scm")}
```

10. A Scheme base

```
; 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

```
(define (get plist key)
  (let ((v (memq key plist)))
    (if v
      (cadr v)
      #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

```
(define (list-find-key lst key)
  (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

```
(define (lfk lst key count)
  (if (null? lst)
    -1
    (if (eq? (car lst) key)
      count
      (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

```
(define (list-set! lst idx val)
  (if (zero? idx)
    (set-car! lst val)
    (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

```
(define (delete! lst key)
  (let ((idx (list-find-key lst key)))
    (if (< idx 0)
      lst
      (if (= idx 0)
```

```
        (set! lst (cddr lst))
        (let ((bef (del-seek lst (- idx 1)))
              (aft (del-seek lst (+ idx 2))))
          (set-cdr! bef aft))))
    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

```
(define (del-seek lst idx)
  (if (zero? idx)
    lst
    (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

```
(define (get-alist lst key)
  (let ((item (assq key lst)))
```

```
    (if item
      (cdr item)
      #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

```
(define (pairlis a b)
  (if (null? a)
    '()
    (cons
      (cons (car a) (car b))
      (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

```
(define (set-alist! lst key val)
  (let ((item (assq key lst)))
    (if item
      (begin (set-cdr! item val) lst)
      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

```
(define (fact n)
  (if (<= n 1)
    1
    (* 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

```
(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