This manual is Copyright 1987, 1990, California Institute of Technology, and is offered under the GNU Public License Version 1, Feb 1989.

NAME

viewmod — interface to the View program

SYNOPSIS

$search '/lib/local/viewmod', '/lib/local/numex'$
import viewmod;

DESCRIPTION

Viewmod is a module of procedures for implementing new commands in the *view(1)* program. To use it, you must search and import `viewmod` as shown above. Note that the file `viewmod.code` file contains only import text; to use this module, you must either perm `view` before your own module, or dynamically load the module with *View*'s *use* command.

Your module must export a procedure of the form:

```
procedure mytoolname_viewinit;
```

where *mytoolname* is the name of the tool, containing no underscores. This name does not need to correspond to the name of the module or code file; it is the name used by the second (optional) argument of the *use* command.

This procedure normally just calls some *viewmod* procedures to add to the *View* command table, and returns. When one of your commands is later used, *View* will call the appropriate procedure in your program.

If your module exports another procedure of the form:

```
procedure mytoolname_exit;
```

then this procedure will be called when *View* is exiting.

INITIALIZATION

A typical `_viewinit` procedure contains one call to `v_addcmd` for each new command implemented, plus extra calls to store additional information about those commands. For example, the following was taken from the built-in commands module:

```
v_addcmd('list', dolist, 'list', 'List all curves in memory');
   v_addhelp('list');
   v_addhelp('list <curve> <curve> <curve>');
   v_addhelp('  List information about curves, or a list ...');
   v_addhelp('  Curves with compatible bases are listed ...');
   v_addalias('ls');
v_addcmd('list*', dolist, 'list <curve>', 'Describe a curve');
```

The first call establishes a new command called `list`. If the user enters a command line whose first word is "list," *View* will strip that word from the line and call the procedure:

```
procedure dolist(buf : newasm_str255);
```

where `buf` contains the arguments portion of the command, if any. When this procedure returns, *View* goes on to read another command from the user.

If the user gives a `?` or `help` command with no arguments, the third and fourth arguments to `v_addcmd` will be written out to describe this command. If the user asks for help specifically about this command, then the text described by `v_addhelp` is written out. The `v_addhelp` procedure appends information to the command established by the most recent `v_addcmd`. The standard help format has one line for each variation of command syntax, then lines indented by two spaces describe how the command works.

The `v_addalias` procedure adds an alternate name for a command. For example, `list` and `ls` are both acceptable names, and the `dolist` procedure has no way to tell which name was used. Alias names must be typed in full, but the "main" command name may be abbreviated to any unambiguous prefix. A command may have any number of aliases.

The second call to `v_addcmd` in the above example does not define a real command; if the name given for a command ends in an asterisk, the name is not recognized in user command lines, but the two help strings are printed. Note that the procedure name must be given but is never used. Calls to `v_addhelp` or `v_addalias` after this call would make no sense.

If both help strings are blank, the command does not show up in the help list. If the strings are `'*'` and `''`, respectively, then the command name is listed at the end of the help list. In both cases, `v_addhelp` should be used in case a user needs to know more information about the command.

If you are writing a View tool for your own use, it may not seem important to include help. But please make the effort, because chances are someone else will eventually use your View tool.

```
v_addcmd('basis', dobasis, '*', '');
   v_addhelp ...
v_addcmd('ylimits', doylimits, '', '');
   v_addhelp ...
```

In these examples, the `basis` command does not get a full line in the help list, but still must be mentioned. The `ylimits` command does not need to be mentioned at all, because the entry for `xlimits` mentions it.

If a call to `v_addcmd` specifies a name that already exists, the old command definition is overwritten. You can use this fact to redefine built-in commands if you wish.

It is also reasonable for the `_viewinit` procedure to call `v_addinterp`, to add new interpolation functions, and `v_addfuncs`, to add new functions for use in expressions.

DATA STRUCTURES

The *View* program has three major data structures: vectors, curves, and bases. A vector is a pointer to an array of reals:

```
v_vectorarray = array [1..1000000] of real;
v_vector = ^v_vectorarray;
```

The context in which a vector is used determines the actual length of the vector, and the programmer is responsible for accessing only those elements that have actually been allocated.

A curve is a record describing one curve in *View*'s memory. Every curve has a vector of "y" values, a pointer to a basis that describes the "x" values, a name and a units string, plus a bunch of other fields that are used for special purposes.

```
v_curveptr = ^v_curverec;
v_curverec =
```

```
record
   next : v_curveptr;
   name, units : na_strptr;
   base : v_baseptr;
   vec : v_vector;
   expr : na_strptr;
   exprtime, chgtime, checktime : integer;
   nex : ne_nexptr;
   used, usedy, flag, flag2 : boolean;
   next2, next3 : v_curveptr;
   dest : na_strptr;
   style, astyle : integer;
   xval, yval : real;
end;
```

All curves are linked by their `next` pointers into a list headed by the global variable `v_curvebase`. Each curve has a unique `name` string; names are case-insensitive. The units string may be blank to indicate that units are not important for this curve. Both of these "strings" are really pointers to variable-length strings. You should use `strnew` and `strchange` from `newasm` to allocate or reallocate these strings; in practice, *viewmod* handles this.

The vector `vec` is points to the values of the curve. The length of this vector is determined by the basis. Each value of this vector must be either a valid number, or the constant `v_badvalue` which marks missing or "undefined" values. This value is shown as "x" when the curve is written out.

The `base` field points to a basis record:

```
v_baseptr = ^v_baserec;
v_baserec =
   record
      next : v_baseptr;
      len : integer;
      units : na_strptr;
      vec : v_vector;
   end;
```

Bases are also linked together, headed by the global `v_basebase`. The `len` field is a positive integer that is the number of values in the vector `vec`, and in any curve that uses this basis. Basis vectors are always sorted into increasing order, with no duplicated or "undefined" values.

For constants (as opposed to curves composed of vectors of data), the `base` and `vec` pointers are `nil`, and the `yval` field contains the value of the constant. Constants are never equal to `v_badvalue`.

For computed curves or constants (as created by the `==` command in *View*), the `expr` pointer is non-nil and points to the expression in string form. The various "time" fields, and the `nex` pointers, are used by *View* to keep track of computed curves and to determine when they must be recomputed.

PARSING

The `strword` procedure from `newasm` may be used to strip words off of the command line, but *viewmod* provides some more specialized parsing procedures:

`procedure v_exstrword(var buf, wrd : string);`
This procedure is equivalent to `strword`, except for a more complex definition of words. Words are terminated by spaces, as in `strword`, but also by any of the characters `:`, `=`, `@`, `[`, `]`, `,`, or `)`. Words always contain balanced parentheses, so that the above characters are not terminators between parentheses. Also, anything

between " or ' quotation marks is taken verbatim. Exactly one comma following the string will be eaten, plus all surrounding blanks. If the string is followed by `:=`, the `:` is eaten so that `:=` may in general be used as a synonym for `=`.

For example, the line:

```
this is , , a  ( t")"est of ) " (words' " etc"("etera
```

would parse into the following words:

```
'this'
'is'
''
'a'
'( test of )'
'" ('words' "'
'etc"("etera'
```

The procedure `v_strword` is just like `v_exstrword` except, if the word began and ended with a quotation mark, the quotes are removed. So, `v_strword` would have parsed the example line as shown above except that penultimate line would have been

```
' ('words' '
```

(In true Pascal syntax, this would be `' (''words'' '`.)

Note that `v_exstrword` will never skip over a terminator character such as `=` or `[`. For example, a parser for a command like

```
tweak curve1 curve2 curve3 = curve4 [0:10]
```

might look like

```
repeat
   v_strword(buf, curvename);
   { save curve name }
until curvename = '';
v_needsep(buf, '=');
v_strword(buf, curvename);
{ save curve name }
if strbegins(buf, '[') then
   begin
      strdelete(buf, 1, 1);
      v_exstrword(buf, lowerlimit);
      v_needsep(':');
      v_exstrword(buf, upperlimit);
      v_needsep(']');
   end
else
   { use default limits }
```

The `v_needsep` procedure checks that `buf` begins with the specified character. If so, it removes that character with `strdelete`. If not, it prints an appropriate error message and aborts the command by doing an `escape(-1)` which is trapped by *View*'s outer loop.

In practice, there are procedures provided for parsing curve names and expressions more conveniently.

function v_findcurve(name : na_str255) : v_curveptr;
This procedure looks up a curve name and returns a pointer to the names curve, or `nil` if the curve was not found.

```
procedure v_curvelist(var buf : string; var cbase : v_curveptr;
                      mode : v_clmode);
procedure v_ncurvelist(var buf : string; var cbase : v_curveptr;
                       mode : v_clmode);
```

These procedures parse a list of names from string `buf` using `v_strword`. These names may include wildcard characters. They create a list among the curve records which is headed by `cbase` and linked by the `next2` pointers. In addition, the `used` flags are set in all curves named, and cleared in all others. The records on the `cbase` list will be in the same order as the names were specified in the command line; for a given name that includes wildcards, elements are in the same order as they appear in the global curve list.

In the case of `v_ncurvelist`, any computed curves whose dependent curves have been changed will be recomputed. (In other words, `v_needcurve` is called on all curves found.) If any curve gets an error while being recomputed, a message is printed as usual and that curve is not added to the list. In the case of `v_curvelist`, the curves are not recomputed and so the caller should not depend on their values being up-to-date. (For example, the built-in `rm` (remove curves) command uses this procedure, since it is working with the curves themselves, not their values.)

The `style` and `astyle` fields are set to 0 and 1, respectively. The `@` syntax may be used to change these, as for the built-in `plot` command. This processing is done only by `v_ncurvelist`.

The `mode` argument must be one of the values `v_clopt`, which allows zero or more curves (and returns `cbase = nil` if there were zero matching curves), `v_clsome`, which allows one or more curves and aborts with an error message if there are no matches, or `v_clone`, which requires exactly one matching curve. Note that in the case of one curve, `cbase` is always a pointer to that curve.

When this procedure returns, either `buf` will be blank, or it will begin with a terminator character.

```
procedure v_desteqsrc(var buf : string; var cbase : v_curveptr;
                      mode : v_clmode);
```

This procedures parses lines of the form `<dest> = <src>`, where the curve names may contain wildcards. If the `mode` is `v_clopt`, and the first curve name is not followed by an `=` sign, then the name is interpreted as `<src>` and `<dest>` is blank. If the `mode` is `v_clone`, then both curves must be specified.

This returns a curve list as for `v_ncurvelist`, headed by the pointer `cbase`. If a destination name was specified, then for every curve on the list, the field `dest` is a pointer to the matching `<dest>` string. For example, when parsing `foo*=bar*`, all curves whose names begin with `foo` will be on the list, and their `dest` strings will be the same as their `name` strings except with `bar` in place of `foo`. If any of the `dest` strings are duplicated, or blank, then the command is aborted with an error message. No attempt is made to look up the `dest` names.

For curves not on the list, and when no destination name was given, the `dest` fields are `nil`.

procedure v_desteqsrc2(dest, src : na_str255; var cbase : v_curveptr);
This procedure does the wildcard processing of `v_desteqsrc`, without the parsing. The string `src` must be a curve name, possibly with wildcards, and `dest` must be either blank or another name with matching wildcards.

```
function v_parsevalue(wrd : na_str255; var r : real) : boolean;
function v_parseinteger(wrd : na_str255; var i : integer) : boolean;
```

These functions interpret the string `wrd` as an expression (see *numex(3)*), and attempt to evaluate that expression. If the expression contains an error or evaluates to a string, a message is written out and the function returns `false` without modifying its argument. If all goes well, the function returns `true` and the result is stored in `r` or `i`. If `v_parseinteger` finds a real number, it rounds it to an integer. Normally, the `wrd` string has been parsed by `v_exstrword`.

If (non-constant) curve names are mentioned in the expression, their "values" will be unpredictable. *View* accepts curve names in such expressions only so that user-defined functions might take curve names as arguments.

function isequal(r1, r2 : real) : boolean;
This function compares two real numbers and returns true if they are equal within 0.001%. *View* always uses `isequal` when comparing real numbers. Note that this function's name does not begin with `v_`.

WRITING MESSAGES

The *View* program keeps a log file of the session. The following procedures may be used to write into this file:

procedure v_logwrite(msg : na_str255);
This procedure writes the string `msg`, prefixed by a `#` character, to the log file. If the log file could not be opened, nothing is done.

procedure v_loginput(msg : na_str255);
This is the same as `v_logwrite` except that it does not prepend the `#` character. This is intended for logging input from the user. The idea is that a log file can be read back in with the `source` command all except the logged input will be treated as comments.

procedure v_logwriteln(msg : na_str255);
This just does a `writeln(msg)`, followed by a `v_logwrite(msg)`. It should be used for all screen output that should also appear in the log.

procedure v_closelog;
This procedure closes the log file. The file will be reopened for appending as soon as it is needed. You should call this procedure before doing a shell escape or other operation that interferes with open files.

procedure v_halt;
procedure v_fail;
These procedures abort the command with escape codes 0 and -1, respectively. In general, `v_halt` just prematurely returns from the command procedure, emulating the C-language `return` statement, and `v_fail` returns with an error indication. If a command that appears in `source`'d file calls `v_fail`, that source file is aborted.

procedure v_failmsg(msg : na_str255);
This procedure writes the message on the screen in red text, adds it to the log, and calls `v_fail`. It is the normal way for commands to signal errors.

procedure v_failneerr(err : ne_errorkind);
procedure v_needcurve;
procedure v_nosuchcurve(name : na_str255);
procedure v_cantcombine;

procedure v_notvector;
procedure v_badunits;
These procedures call `v_failmsg` in various standard ways. `V_failneerr` writes a *numex(3)* error message. The others write the following messages, respectively:

```
Need a curve name
No such curve as <name>   (or like v_needcurve if name = '')
Can't combine curves with different bases
Curve must be a vector of values
Inconsistent units
```

HANDLING DATA

The following procedures help manage vectors, curves, and bases.

procedure v_newvector(var vec : v_vector; len : integer);
procedure v_disposevector(var vec : v_vector; len : integer);
These procedures allocate or dispose a vector. The `len` argument is in units of vector values, not bytes.

procedure v_makecurve(var cp : v_curveptr; bp : v_baseptr;
vec : v_vector; units, name : na_str255);
This is the basic procedure for creating new curves. The `bp` argument is a pointer to a basis, and `vec` is a pointer to a vector of the same length as the basis vector. (The `vec` pointer must not point to a vector already used in some other curve or basis.) The `name` and `units` strings are copied into the curve record. If a curve with the same name already exists, it is deleted and replaced by the new curve, and a message is written to the screen. The `expr` and related fields are set to `nil`, and `xval` and `yval` are set to zero.

The argument `cp` is pointed to the newly-created (or overwritten) curve.

procedure v_addcurve(len : integer; xvec, yvec : v_vector;
xunit, yunit, name : na_str255);
This is a higher-level interface to `v_makecurve`. If the values in `xvec` are out of order or contain duplicates, the vectors are sorted in parallel. If there is a basis existing that matches `xvec` and `xunits`, it is used; otherwise, a new basis is created. Then, `v_makecurve` is called to create the curve.

Unlike `v_makecurve`, this procedure always copies the vectors `xvec` and `yvec` before storing them. The original vectors are not modified (even during sorting) or deallocated. For example, the built-in `splice` operation on curves `cp1` and `cp2` could be implemented by:

```
v_addcurve(cp1^.bp^.len, cp1^.vec, cp2^.vec,
           cp1^.units^, cp2^.units^, name);
```

If either of the vectors you supply were allocated specially for the occasion, you must deallocate them afterwards with `v_disposevector`.

There are two global vectors, `v_tempxvec` and `v_tempyvec`, which may be used for building up temporary data. You should never call `v_newvector` or `v_disposevector` on these vectors; they are allocated automatically. Also, you should not assume they will remain valid between calls to your commands, or across a call to `v_docommand`.

procedure v_addcurveconst(val : real; yunit, name : na_str255);

This procedure creates a constant-valued curve. (For example, the *View* command "a=2*3" would create a constant-valued curve a with value 6.) This procedure just calls v_makecurve with a nil basis and vector, and then stores val in the curve record returned.

procedure v_addcurveexpr(expr, name : na_str255);
This procedure creates a computed curve. Note that the expression will not actually be evaluated until the curve is next accessed. For example, the expression may include names of curves that have not yet been defined. Also, note that there is no yunit argument; units are not determined until the expression is parsed and evaluated. (If you wrote a==b, for example, the units of a depend entirely on the units of b.)

procedure v_deletecurve(var cp : v_curveptr);
This procedure simply deletes the curve pointed to. The argument cp is changed to nil. The curve record, and all associated strings and vectors, are disposed. There is no similar routine for the rename or copy commands, but you can access these operations through the more general v_docommand function.

procedure v_change(cp : v_curveptr);
This procedure marks a curve as changed. Next time any computed curve that depends on this one, directly or indirectly, is accessed, that curve will be recomputed. The v_makecurve procedure calls this, as do various other *viewmod* operations.

procedure v_needcurve(cp : v_curveptr);
This procedure checks the curve cp and recomputes it if necessary. It should be called whenever a curve's values, or units, or information about its basis, are needed. If an error occurs, the procedure will call v_fail. This is called by v_ncurvelist, v_desteqsrc, v_desteqsrc2, and v_checkneeds.

procedure v_stretchtempvecs(len : integer);
This procedure makes sure that the temporary vectors have been allocated to at least the specified length. If not, they are reallocated and the original contents copied.

function v_isvalid(cp : v_curveptr; n : integer) : boolean;
This procedure takes a curve, cp, and an index n between one and the length of the curve, and checks whether that value is invalid (i.e., equal to v_badvalue). For compatibility with the future, this should always be used rather than an explicit comparison.

procedure v_checkcurve(cp : v_curveptr);
This procedure checks all values of a curve and aborts with an error message if any of them are undefined, i.e., equal to v_badvalue.

INTERPOLATORS

The *View* program has a fairly general curve-interpolator mechanism. This is the mechanism used for the built-in map command. Users can use interpolation in their own commands, and can also define their own interpolation functions.

```
function v_parseinterp(var buf : string) : v_interpolator;
procedure v_disposeinterp(var int : v_interpolator);
function v_interp(var int : v_interpolator; cp : v_curveptr;
                  x : real) : real;
```

These procedures handle curve interpolation. The first procedure parses an interpolation specifier, as in `[linear]`, and returns a record identifies the interpolation type. If the string `buf` did not begin with a `[` character, it returns the code for linear interpolation. Otherwise, it deletes the bracketed specifier from `buf`.

The second procedure disposes an interpolator record when you no longer need it.

The third procedure computes the interpolated (or extrapolated) value of curve `cp` at x-value `x`, using interpolation mode `int`. The default mode does linear interpolation between points, and return `v_badvalue` beyond the first or last point. The built-ins module defines a second interpolator, `lextr`, which also does linear extrapolation.

procedure v_addinterp(name : na_str255; proc : v_interpproc;
help : na_str255);

This procedure adds a new interpolator to the system. The `name` is a string as would appear in brackets in a `map` command. The `proc` is the procedure that computes the interpolation function. The `help` string will be printed by future `interp` commands to describe this interpolator.

The `proc` procedure is of the form:

```
procedure myinterp(cp : v_curveptr; x : real; user : na_longword;
                   var val : real);
```

Every time an interpolated point is needed, this procedure is called with `cp` pointing to the curve in question, and `x` equal to the x-value at which the curve's y-value is needed. The procedure should compute the value and return it in `val`. It is legal to return `v_badvalue` in `val`. The `user` parameter may be ignored unless you are using the features described below.

This procedure should make no assumptions about the order or nature of x-values requested. If this is inconvenient, it may perform caching, as follows: if the curve pointer `cp` is the same as for a previous call, and the `user` value is the same, then a cached interpolation may be used. The parsing procedure should set `user` to a (somehow) unique value on every call. *View*, and other callers of `v_interp`, should guarantee that a given interpolator as returned by `v_parseinterp` will never be used more than once per curve, or, more precisely, once `v_interp` is called, the values of that curve should no longer be changed for as long as that `v_interpolator` is used.

procedure v_addintparse(pproc : v_pinterpproc; dproc : v_dinterpproc);

This procedure specifies a parsing procedure for the most recently added interpolator. The procedure arguments should be of the form

```
procedure mypinterp(var buf : string; var int : v_interpolator);
procedure mydinterp(var int : v_interpolator);
```

where `buf` is a string containing the remainder of the line after the interpolator's name was parsed, and `int` is an interpolator record describing the interpolator in question.

The `pproc` procedure is called by `v_parseinterp`. Normally, the `buf` string will begin with a `]` character and should not be modified, but interpolators may specify additional arguments to the interpolation. The parsing procedure should parse these arguments and store them in `int.user`. This field is of type `na_longword`, which is a variant record defined by *newasm(2)* that contains various interpretations of a 32-bit storage location. For example, `user.i` is an integer, or `user.p` is an anyptr that could point to a record of additional information.

The `dproc` procedure is called by `v_disposeinterp`. If `pproc` allocated any extra information (pointed to by `user.p`), this procedure should deallocate that information.

DEFINING FUNCTIONS

The *View* program also gives you the opportunity to define your own functions, to be used anywhere expressions are allowed.

procedure v_addfuncs(proc : v_funcproc);
This procedure records the procedure `proc` in a list of function establishers. The procedure `proc` will be called every time the *numex(3)* symbol table descriptor, `v_nedesc`, must be recreated (e.g., when the list of curve names changes).

The function establisher should call the various `numex` facilities for defining new functions, constants, and/or variables, as described in that module's `man` page. For defining real-value functions of one argument, *View* defines the following shorthand:

procedure v_realfunc(name : na_str255; proc : ne_prreal);
This procedure, to be called from a function establisher, creates the function `name`, handled by procedure `proc`, which might be of the form:

```
procedure sqrtproc(nex : ne_nexptr; var res : real);
   begin
      res := sqrt(ne_reval(nex^.p1));
   end;
```

(though, of course, the `sqrt` function is already built-in). For your information, the `v_realfunc` procedure is defined as follows:

```
procedure v_realfunc(name : na_str255; proc : ne_prreal);
   var
      l1 : na_strlist;
   begin
      strlist_add(v_nedesc.symtab, l1, strupper(name));
      ne_makerealfunc(l1^, ne_real, ne_notype, proc);
   end;
```

procedure v_scrapsymtab;
procedure v_buildsymtab(t : integer);
Your own commands may use *numex(3)* to parse and evaluate expressions. You should use the `v_nedesc` descriptor so that all *View*'s built-in functions are recognized. Before using `v_nedesc`, you must always call `v_builtsymtab` to set it up. The `t` parameter describes in what context the symbols will be used. Currently defined values are 0, for simple expressions as in `v_parsevalue`, and 1, for curve expressions as in the *View* assignent command. (At present, there is no difference between these two modes, but they are supported in case features should be added in the future.)

`V_buildsymtab` will not rebuild the symbol table unless the `t` value is different that from the last call, or unless `v_scrapsymtab` has been called. *View* calls `v_scrapsymtab` whenever the curve list changes.

This procedure creates three symbols for every existing curve (including constants); their names are `<curve>`, `<curve>_y`, and `<curve>_x`. They are all declared as variables (type `ne_rp`), and pointed to the `yval` (in the first two cases) or `xval` (in the latter case) of the relevant curve record.

procedure v_checkcurvename(var name : na_str255; var cp : v_curveptr;
var mode : char);

This procedure looks up a curve name and stores a pointer to that curve in `cp`. If the name is not found, and the name has a suffix `_c` for some character `c`, then the curve without the suffix is looked up, and the suffix is returned in `mode`. If there is no suffix, `mode` is not changed.

This procedure is useful if a user-written function needs access to a curve's name (rather than its value). For example, if the first argument is to be interpreted as a curve name, the function can write:

```
if nex^.op <> ne_rp then         {error - not a variable!}
v_checkcurvename(nex^.rps^.s, cp, mode);
if cp = nil then                 {error - not a curve name!}
```

which will locate the curve's record. The global variable `v_arithidx` contains the index into the curve which is currently being worked on, and `v_arithbase` is the basis of the curves in the expression. These are enough to provide a wide variety of functionality.

procedure v_checkneeds(nex : ne_nexptr);
This procedure checks the compiled expression `nex` and, for every curve named in that expression, calls `v_needcurve`. It should be done on all compiled expressions, not just "curve expressions," because any expression may want to mention a constant-valued curve.

ARITHMETIC ON CURVES

This section describes the (rather kludgey) system for performing arithmetic on curves, as is done by the built-in assignment command.

```
procedure v_curveexpr(wrd : na_str255; var nex : ne_nexptr;
                      var bp : v_baseptr; var cbase : v_curveptr;
                      var units : na_str255);
```

This procedure parses the string `wrd` as an expression, using the `ne_compile` procedure of *numex(3)*. The resulting expression tree is returned in `nex`. In addition, the tree is scanned for curve names and those names are looked up. On return, `bp` will point to the basis of all the curves mentioned in the expression, and `cbase` points to a `next3`-linked list of those curves. (Both are `nil` if there were no curve names; only `bp` is `nil` if there were only constant-valued curve names.) The `used` and `usedy` flags for the various curves will be true if `curve_x` and `curve` (and/or or `curve_y`) were mentioned, respectively. The string `units` contains the units of the resulting curve, formed by combining the units of the various constituent curves. Finally, `v_checkneeds` is called on the expression.

```
procedure v_evaluate(nex : ne_nexptr; bp : v_baseptr;
                     cbase : v_curveptr; vec : v_vector);
```

This procedure evaluates the expression `nex` at each point in the basis `bp`. Normally, the `nex`, `bp`, and `cbase` arguments are the values returned from `v_curveexpr`. The resulting values are stored in vector `vec`, which must have been allocated to at least the size of `bp`. (You may use `tempxvec` or `tempyvec` for this purpose.)

If any errors occur, the appropriate error message is written and the corresponding entry in `vec` is made undefined. In addition, if any of the constituent curves were undefined at a given point, the expression is not evaluated at that point and the resulting value is undefined.

As an extended example, the command `<dest> = <src>` is implemented as follows in the built-ins module:

```
v_curveexpr(src, nex, bp, cbase, units);
if bp = nil then
   v_addcurveconst(ne_revaluate(nex, v_nedesc), units, dest);
else
   begin
```

```
            v_newvector(vec, bp^.len);
            v_evaluate(nex, bp, cbase, vec);
            v_makecurve(cp, bp, vec, units, dest);
         end;
      ne_dispose(nex);
```

COMMANDS

The following procedures allow you to execute arbitrary *View* commands recursively:

function v_docommand(buf : na_str255) : boolean;
This procedure executes the command `buf`, as if it were typed in at the keyboard or read from a source file. If it succeeds (i.e., the command name was recognized, and the associated command procedure returned normally or called `v_halt` to exit), then this procedure returns true. If any errors occurred, it returns false.

function v_sourcefile(fn : na_str255; trylib : boolean) : boolean;
This procedure reads commands from the file named. A `.text` extension is added by default. If the file is not found, or if any of the commands returned an error condition, `v_sourcefile` returns false. If everything went well, it returns true.

If `trylib` is true, this procedure tries first the current directory, then the directory `/lib/view`. If `trylib` is false, only the current directory is tried. (There is no difference if the name includes an explicit directory path.)

function v_use(fname, tname : na_str255) : boolean;
This procedure loads a new View tool. It is equivalent to a "`use <fname> <tname>`" command.

BUGS

Arithmetic should be done on whole curves, not elementwise.

FILES

`/src/util/view/view.text`   `Source file for viewmod and View`

SEE ALSO

view(1) numex(3) regex(2)

AUTHOR

Dave Gillespie [DAVE]