Programming in View

This document describes the programming language embedded in the View system. The View programming language has elements borrowed from conventional programming languages like C and Pascal, and from Unix shell languages. Though not intended as a high-performance language, it is fast enough for many control applications. The language allows authors to write new View commands similar to the existing ones; these commands can be simple enough even for novice users.

When View is started it normally reads commands first from the file `/lib/view/view.cnf`, then from `view.cnf` in the current directory, then finally from the keyboard. A transcript of the session is written to the file `view.log`. There are various command-line options for omitting the configuration files, reading commands from a user-specified file, and changing or disabling the log file. These command-line options are described in the manual page for View.

One especially interesting View option is `-c`. The remainder of the arguments after `-c` are parsed as a View program file name followed by argument words. The number of argument words is stored in a numeric View variable `argc`, the words themselves are stored in string variables `arg1`, `arg2`, etc., and then commands are read from the program file. The Chipmunk Shell recognizes the `.view` extension and automatically runs View on such files when the user executes them. For example, you can write a View program in `foo.view`, place that file somewhere in your search path, then run the program from the Shell just by typing `foo`. This is especially handy as the Chipmunk Shell itself has only very limited programming facilities.

Concepts

The three major kinds of entities in the View system are *commmands, variables,* and *parameters.* Commands are implemented by combinations of procedures written in View, and imported Pascal procedures. Variables contain numbers, strings, or vectors of x-y data pairs ("curves"). Parameters are like predefined variables with special properties. There are separate name spaces for each, so you could have one command, one variable, and one parameter each named `foo` and they would not conflict.

Command names are case-insensitive and may be abbreviated to enough letters to make the command unique. (For example, if there were the commands `plot`, `print`, and `plink`, they could be abbreviated as `plo`, `pr`, and `pli`, respectively.) Certain "control" commands, such as `while` and `endwhile`, must always be spelled out in full. Also, usually when you are referring to a command name in another command (such as `undefine`) you must spell out the name in full.

Variable names are case-sensitive: `a` and `A` are different variables. Also, abbreviations for variable names are not supported. Variable names may contain nearly any characters, but it is best to use only letters, digits, and the characters "`_`" and "`.`" and "`#`", and to begin them with letters. (When variable names are written in arithmetic expressions, only these characters are allowed.) Variables are freely typed, so that `a` may contain a number at one time, and later be changed to a string or curve. There is no need to declare variables.

Parameter names are like commands: case insensitive and able to be abbreviated. Parameters have fixed types, and depending on the parameter may have special properties. For example, if you change the `logfile` parameter to name a new file, this causes View to close the old file and open the new file automatically.

View procedures can have local variables, parameters, and commands. When the procedure returns, the previous values of its locals are restored automatically. Note that all scoping in View is *dynamic,* not *static* as in Pascal. For example, if procedure `x` creates a local variable `v`, then calls `y`, which uses `v`, then `y` will use the local, not global, `v`, even though that variable was not actually declared in `y` itself. As another example, if procedure `x` makes `logfile` a local parameter and changes it to name a different file, then that log file will also be used by all procedures called by `x`. When `x` returns (normally or abnormally), the log file name will be restored to its original value.

View is an interpreted language. That is, there is no "View compiler." You just write a program and then run it in View. The View system is broken up into the "Reader," which actually reads commands from the keyboard or from a file, and the "Expander," which interprets various "`$`" notations that will be explained later, and the "Interpreter" that actually executes each command. Each of these stages looks for characteristic symbols such as "`##`" and "`$`". What they do with them varies; one universal rule is that anything between quotes (`' '` or `" "`) is never given any special interpretation.

The Reader

View is a line-oriented language. That is, normally there is exactly one command per line and thus there is no need for all the semicolons you see in free-form languages like Pascal. The `;;` symbol lets you put two commands per line. Actually, when the View reader sees `;;` it splits the input up and the rest of View thinks you really entered each command on its own line. Similarly, you can split one command over several lines with the \\ symbol:

```
readnum a b ;; print a '+' b '=' a+b
```

can also be written as

```
readnum a b
print a \\
      '+' \\
      b \\
      '=' \\
      a+b
```

When the View reader sees a line that ends in \\, it reads the following line and appends it onto the current line, minus the \\, and with one separating space. The total length of any command must be less than 255 characters no matter how many lines it is written on.

The other thing the Reader does is strip off comments. A comment is anything on a line following a "`##`" mark. (If the comment is the only thing on a line, you can use just one "`#`" if you want.) For example:

```
# Now print the answer
print a \\     ## the first variable
      '+' \\
      b \\     ## the second variable
      '=' \\
      a+b      ## the sum
```

That's all there is to the Reader stage. The above transformations occur just once whenever you type a command line at the keyboard, and for every line that is read from a View program file. If you give the commands:

```
view> a=5;; b=7
view> history
```

the `history` command will list the `a=5` and `b=7` as if you entered them on different lines.

The Expander

The Expander is the stage of the View interpreter that processes each View command before the Interpreter executes it. It scans the command line for "`$`" characters not enclosed in quotes and interprets them according to the rules described below. If you are not interested in all the gritty details, you can skip this section and come back to it later.

The simplest expansion is just a `$` followed by a variable name. This expands into the value of the variable expressed as a string. If the variable currently holds a number, it expands into, say, '`1.3E-7`'. (If the variable holds a curve, the expansion will not work.) For example,

```
i = 3
j = 6
a$i = a$j
```

Here, the third line will expand to

```
a3 = a6
```

before it is executed. If the expansion must be followed immediately by a character that would be interpreted as part of the variable name, you must enclose the variable in curly braces:

```
a${i}b$j = a${i+1}b$j      ## where i=3, j=6
```

expands to

```
a3b6 = a4b6
```

This example also illustrates that you can actually write any arithmetic expression between curly braces; it will be evaluated, converted to a string if necessary, and used in the expansion.

You can modify the effects of the expansion with "format characters" separated from the expression by a colon:

```
view> a = '  Hello  '
view> echo *$a*
*  Hello  *
view> echo *${a:u}*     ## Convert to upper case
*  HELLO  *
view> echo *${a:l}*     ## Convert to lower case
*  hello  *
view> echo *${a:t}*     ## Trim leading/trailing spaces
*Hello*
view> echo *${a:q}*     ## Quoted
*'  Hello  '*
```

The `:q` format surrounds the replacement string with quotes so that it can be read back in as an expression. If the string contains apostrophes, it automatically chooses double-quotes instead; if the expression contains both kinds of quotes, it pairs quotes inside the string:

```
view> a = 'He said, "that''s great"'
view> echo $a
He said, "that's great"
view> echo ${a:q}
"He said, ""that's great"""
```

Another handy format is `:n`, which transforms the replacement string so that it contains only characters which can occur in variable names; this transformation is guaranteed to be unique for any real number:

```
view> echo foo${35:n}
foo35
view> echo foo${-23.6e+004:n}
foo_23.6e004
```

So, you can use `:n` format to create arrays indexed by real numbers in the form of groups of similarly-named variables. In fact, the "`#`" character is valid in variable names, and is intended as an "array index separator." Where in Pascal you would say `a[i,j]`, in View you would say:

```
a#${i:n}#{j:n}
```

True, it's a little more awkward, but it gets the job done!

If there is a second "`:`" character between the curly braces, it signifies a default replacement string:

```
view> readln a
a? hello
view> echo ${a:q:goodbye}
'hello'
view> a = ''
view> echo ${a:q:goodbye}
goodbye
```

If the expansion evaluates to an empty string, the Expander uses the default string instead. Note that the default string is taken exactly as-is, regardless of whatever formatting characters appear. But, the Expander applies itself to the default string so that you can use nested expansions:

```
echo ${a:q:${b:q:'goodbye'}}
```

will echo `a` in quotes if `a` is non-blank, otherwise `b` in quotes if `b` is non-blank, otherwise `'goodbye'` if both `a` and `b` are blank.

The `:e` formatting character traps errors while evaluating the expression. For example, ordinarily `$a` will fail if there is no variable `a` defined, but `${a:e}` will simply expand to a blank string if `a` is undefined. This is handy in conjunction with defaults:

```
view> rm *         ## Delete all active variables
view> i = -3.4     ## Define one variable
view> echo a${i:ne:0}, a${j:ne:0}
a_3.4, a0
```

The `:v` formatting character says that the thing to the left of the `:` is not an expression, but is to be taken verbatim:

```
view> echo *${hello:v}*
*hello*
view> echo *${a${i:n}:qv}*
*'a_3.4'*
```

The `:p` formatting character says that the thing to the left of the `:` is a parameter name. For example, if the `logfile` parameter contains the string `VIEW.LOG`, then

```
view> echo ${logfile:pq}
'VIEW.LOG'
```

As a special shorthand, you can use parentheses instead of curly braces to expand a parameter name:

```
view> echo ${logfile:p} is just like $(logfile)
VIEW.LOG is just like VIEW.LOG
view> echo ${logfile:pq} is just like $(logfile:q)
'VIEW.LOG' is just like 'VIEW.LOG'
```

Remember that parameter names allow shorthand: it is probably enough to say `$(log)` instead of `$(logfile)`. But if you are writing a program that is to work under battle conditions, you should spell it out so that it doesn't fail if somebody creates a new parameter named, say, `logjam`.

If the `$` character is followed by anything other than a letter, `{`, or `(`, it simply "quotes" that character to hide it from further expansions. This is most useful for quoting another dollar sign:

```
view> i = 3.14;; echo It costs $$$i
It costs $3.14
```

The only other detail with regard to the Expander is that occasionally the Expander is called at several different times before executing a command, and there is a special way to specify at which of those times you wish the expansion to occur:

```
i = 7
define foo          ## Define a command called foo
   i = 5            ## ... which changes i
   define bar       ## ... and defines bar
      i = 3
      echo ${{{i}}}
      echo ${{i}}
      echo ${i}, $i
   enddef ##bar
   echo ${{i}}
   echo ${i}, $i
enddef ##foo
```

After reading the above definition, View will have defined the command `foo` as follows:

```
i = 5
define bar
   i = 3
   echo 7
   echo ${{i}}
   echo ${i}, $i
enddef
echo 7
echo ${i}, $i
```

and when the `foo` command is executed, it will write

```
7
5, 5
```

and define `bar` as follows:

```
i = 3
echo 7
echo 5
echo ${i}
```

Finally, when you execute `bar` it will print:

```
7
5
3, 3
```

In words, when View reads the body of a `define` statement it runs the Expander on those commands as it reads, and the interpreter will call the Expander again when the defined command executes. An expansion that appears nested within n `define` commands must use $n + 1$ braces or parentheses in order to be expanded immediately; otherwise, it is expanded when the command it appears in is actually executed.

Since nested `define` commands are very rare, it is easiest just to remember that `$a` and `${a}` are expanded when the command is *executed,* but `${{a}}` is expanded when the command is *stored* in the definition of another command.

The Interpreter

After the Reader and Expander have done their job on a command line, the Interpreter figures out what that command is supposed to do.

The first step is to read the first word on the line. For now you can think of a "word" as a sequence of characters terminated by a space, punctuation character like "`=`", or end-of-line. If that word is followed by an `=` sign, the Interpreter assumes this is an assignment statement, i.e., a "`let`" command, and the first word is the name of the variable being assigned. Otherwise, that first word is taken as the command name. View looks up the command name to see its current definition, then takes the appropriate action depending on that definition. If the command is built-in or is defined in a Pascal View tool, the Interpreter just executes the Pascal procedure associated with the command and passes that procedure the remainder of the command line, minus the command name.

For example, if you give the following commands:

```
view> a = 6
view> print a
```

the first is converted to "`let a = 6`" and then the Interpreter calls the built-in procedure for the `let` command (assuming you have not redefined that command!). Then, it calls another built-in procedure for the `print` command.

When you define your own commands in the View language, the interpreter starts to read commands from the definition; when it reaches the end of the definition, it returns to the original source of commands. For example, in:

```
view> define foo
view>> a = $a$a + 1
view>> print a
view>> enddef
view> a = 3
view> foo
view> foo
view> quit
```

the built-in commands that are executed are:

```
define foo
let a = 3
let a = 33 + 1
print a
let a = 3434 + 1
print a
quit
```

and the output will be the numbers `34` and `3435`.

You can also define a command that takes arguments (as, for example, a `foo a` that acts on variable `a`), and you can create multiple definitions for a command with various patterns of arguments. In this case, the Interpreter examines each definition for the command and tries to match the arguments you gave to the argument patterns in the definitions. When it finds a definition that matches, it begins to execute commands from it as above. If it finds no suitable definitions, and there is no Pascal procedure to handle this command, it prints an error message and stops. (Normally, "stopping" entails aborting whatever definitions or command files were in progress and returning to keyboard input mode for the next command.)

See the later section on *Procedures* for more details on argument matching.

As a special case, if you precede a command by the word "`builtin`," then any View language definitions that have been added for that command are ignored—the Interpreter goes directly to the Pascal definition. For example, you can use

```
builtin a = 6
builtin print a
```

if you have added definitions to "`let`" and "`print`." Note that if you have actually *replaced* the built-in Pascal definition for a command, there is no way to get it back. The `builtin` syntax works only if you have *augmented* the command with some code for special cases and you now wish to ignore that code. A good example is shown in the section on *Parameters.*

Assignments

The basic way to create a variable in View is with the `let` command:

```
view> let a = 6
view> print a
6
```

The word `let` is optional, so you can write simply:

```
view> a = 6 ;; print a
6
```

for exactly the same effect.

The righthand side of the " =" sign may be any arithmetic, string, or curve expression. The expression is evaluated and then assigned to the variable on the lefthand side. So, the righthand side may refer to the old value of the variable being assigned:

```
view> a = a + 1 ;; print a
7
```

When assigning curves, the entire curve is computed before being assigned to the lefthand variable. For curve computations, the expression is evaluated once at each point. For example, computing `a*b` where both `a` and `b` are curves will produce the pointwise product of the curves. It is not possible to combine curves with different bases. For more information, see the section on *Curve Manipulation Commands.*

It is also possible to create a *computed variable.* If you use a double equals sign,

```
view> a == 2*b
```

then variable `a` actually remembers the expression `2*b` from which is was computed. If the variables in that expression (i.e., `b`) ever change, the variable `a` will be recomputed automatically.

```
view> a == b+b
view> print a
Variable a: No such name
view> b = 5;; print a
10
view> b = 'hello';; print a
hellohello
```

Note that for "`==`" assignments the expression does not need to be valid until the variable is first *used.* As another example,

```
view> c == a + b
view> a = 1;; b = 2;; print c
3
view> a = 'hi';; b = ' there';; print c
hi there
```

Notice that after the `a='hi'` command, `a` is a string and `b` is a number and so the expression computing `c` is invalid. However, no error message is printed because we don't *refer* to `c` until `b` has also changed to a string.

You can reassign to a computed variable directly with another assignment command. If you reassign with "`==`", you change the expression computing that variable. If you reassign with "`=`", you change the variable back to being a normal, noncomputed variable.

It is not legal to create a loop of dependencies of computed variables:

```
view> a == b
view> b == c
view> c == a
view> print c
Variable c is recursively defined
```

However, it is perfectly alright to create a chain or tree of dependencies:

```
view> a == b+2
view> b == c*3
view> c = 4
view> print c b a
4 12 14
```

Beware that complicated networks of dependencies, especially involving curve-valued variables, can slow down the use of those variables. The View system always displays the message "Recomputing xyz" whenever it is taking time out to recompute a variable, just so that you don't think the computer has gotten stuck.

Expressions

This section is unfinished.

Some interesting functions that may not be obvious:

`cvlen('`*curvename*`')`

Returns the number of data points in the specified curve. Note that the *curvename* is given in quotes; actually, any string expression may be used here.

`cvidx('`*curvename*`',` *index*`)`

Returns the y-value of a given point of a curve, or the x-value if the curve name is followed by "`_x`". The integer *index* must be between one and `cvlen('`*curvename*`')`, respectively. As a more interesting example,

```
foo = 'bar'
len = cvidx(foo)
val = cvidx(foo+"_x", len)
```

which gets the last x-value of the curve named by string `foo`, i.e., of curve `bar`.

`cvrelidx('`*curvename*`',` *delta*`)`

Returns the x- or y-value of a given point of a curve. Intended for use in curve expressions. Using *delta*=0 is just like writing the curve name by itself in the expression. For example,

```
a = b + cvrelidx('b', -1)
```

causes the first element of `a` to be undefined, the second to contain the sum of the first two elements of `b`, the third to be the sum of the second and third elements, etc. Outside of curve expressions this function is not useful.

`if(`*condition*`, then(`*thenexpr*`,` *elseexpr*`))`

Does a conditional computation. If *condition* is nonzero, the *thenexpr* is evaluated and used, otherwise the *elseexpr* is evaluated and used. The kludgey syntax will soon be replaced by something better as View's expression parser is improved.

`rand(`*maxval*`)`
`rand2(`*minval*`,` *maxval*`)`

Generate random numbers. The `rand` function generates an integer between zero (inclusive) and *maxval* (exclusive). The `rand2` function generates a real number in the indicated range.

`min(`*a*`,` *b*`)`
`max(`*a*`,` *b*`)`

Take the smaller or larger of two numbers. Again, these will soon be replaced by version that can take any number of arguments.

`arctan2(`*y*`,` *x*`)`

Compute the inverse tangent of y/x, in the interval $[-\pi, \pi)$.

Variable Manipulation Commands

This section is unfinished.

Curve Manipulation Commands

This section is unfinished.

One command that may be unfamiliar is `poke`:

> `poke` *curvename* `@` *index* `=` *value*

This changes one of the y-values of the given curve to the given value. The *index* must be between one and the number of points in the curve. This command never creates new points in a curve, nor does it modify the x-values of points. To modify x-values you can use `poke` along with the `splice` command.

Control Constructs

There is a set of commands known as *control commands* which affect the flow of control in a View program. They are special in that their names must always be written out in full, and their names must appear explicitly on the command line. For example, if you said:

```
ending = 'endif'
if foo = bar
   print foo
${ending}
```

The above sequence will not work; for example, if it turns out that `foo` $\neq$`bar`, then the Interpreter will try to skip to the matching `endif` command and not find it—for efficiency, while skipping the Interpreter does not use the Expander or understand command abbreviations and aliases.

Some of the control constructs are repetitive. For example, the `while` loop repeats the commands between `while` and `endwhile` as long as some condition is true. If the condition is false, it never executes the commands, but skips directly to the `endwhile`. When you are entering a looping command from the keyboard, the first iteration, if any, will happen as you type it; the commands you type are also being saved away for the later iterations of the loop:

```
view> i = 2
view> while i >= 0
view>> print i
2
view>> i = i - 1;; endwhile
1
0
view> while i >= 0
skip>> dfglhghdgk  ## this is ignored!
skip>> endwhile
view>
```

If you get stuck typing a structured command and want to abort it, just press the [STOP] key. If you are asked if you want to exit the View system, answer *no*. You will return to the command prompt, but any structured commands that were in progress will have been cancelled.

The following sections describe each of the built-in control constructs in turn: `if`, `switch`, `while`, `repeat`, `for`, `try`. (The `define` construct is described later.) You can also define your own commands which act like control constructs, using the `define [block]` option.

All control constructs can be nested. For example, you can have one `while` loop inside another just as in Pascal. When inputting from the keyboard, the prompt grows an extra `>` for every nesting level you are in. However, it does not grow for nested commands that View does not need to keep track of: while skipping over an `if`, View keeps track of nested `if`'s but ignores `while`'s and `endwhile`'s completely.

The `If` *Construct*

The `if` construct in its simplest form checks if some condition is true. If so, it executes the commands in its body. Otherwise, it skips these commands.

```
readnum npig : How many pigeons?
if npig > 10
   echo Too many pigeons!
   npig = 10
endif
print 'Using' npig 'pigeons.'
```

In this example, if the user enters 10 or less pigeons, the Interpreter skips directly to the `print` command. But if the user asks for more than 10 pigeons, the `echo`, assignment, and `print` commands are each done in turn.

The `else` command specifies an alternative:

```
if npig > 10
   echo Too many pigeons! ;; npig = 10
else
   print 'Using' npig 'pigeons.'
endif
```

In this case, either the `echo` and assignment or the `print` are executed, but never both or neither. In all cases, the Interpreter will then go on to the next command after the `endif`.

Often a program must select among several alternatives. You can write such an alternative like this:

```
if npig > 10
   echo Too many pigeons!
else
   if npig < 0
      echo Too few pigeons!
   else
      echo Just right.
   endif
endif
```

but there is a much cleaner form that uses the `elseif` command:

```
if npig > 10
   echo Too many pigeons!
elseif npig < 0
   echo Too few pigeons!
else
   echo Just right.
endif
```

Note that the full structure of an `if` construct is an `if` clause, followed by any number of `elseif` clauses, then an optional `else` clause, and finally `endif`.

The View program always ignores indentation of commands, but it is a good idea to indent as shown above (or according to your own favorite scheme). Without indentation, View procedures quickly become unreadable.

The `Switch` *Construct*

The `switch` construct is a handy way to select among several choices. For example,

```
switch x
   case 1
      print 'one'
      break
   case 2
      print 'two'
   case 3
   case 4
      print 'two, three or four'
      break
   otherwise
      print 'I dunno'
endswitch
echo After switch
```

In general, the Interpreter evaluates the switch expression (in this case the variable `x`), then checks each `case` in turn to see if it matches. If any case matches, it then begins executing commands. If no cases match, it executes the `otherwise` clause. (If nothing matches and there is no `otherwise` clause, it skips straight to the `endswitch`.)

The `break` statement is used to break out of a construct. In a `switch` construct, it jumps immediately to the `endswitch`. So, for example, if `x=1`, the commands executed are:

```
switch x
case 1
print 'one'
break
echo After switch
```

whereas if `x=2`, the commands are:

```
switch x
case 1
case 2
print 'two'
case 3      ## ignored
print 'two, three, or four'
break
echo After switch
```

Note that the combination `break`–`otherwise` is similar to `else` in an `if` construct.

The values following `switch` and `case` may be arbitrary expressions. If the results are numbers, they are converted to strings before being compared. (Ordinarily, this will not make much difference.) This means that `switch` is a handy way to compare a number with a string of digits to see if they represent the same thing.

Normally comparisons are case-sensitive. That is, in

```
switch foo
   case bar
   ⋮
```

where `foo='yow'` and `bar='Yow'`, the expressions will not match. You can specify case-insensitive comparisons as an *option:*

```
switch [ci] foo
   case bar
   ⋮
```

This compares the switch value with the cases without distinguishing capital and lower-case letters. You could get a similar effect by saying:

```
switch ${foo:uq}        ## expands to: switch 'YOW'
   case ${bar:uq}       ## expands to: case 'YOW'
   ⋮
```

but the "`[ci]`" notation is much cleaner.

Instead of saying `case` you can say `pcase` to compare the switch value against a pattern containing wildcard characters. The wildcards are "`*`", which matches zero or more characters, "?", which matches any single character, and, e.g., "[a-zA-Z]" which would match any letter. You can also mix the different kinds of `case`'s in the same `switch`:

```
switch n+m
   case 0
      echo Zero;; break
   pcase *.*
      echo Fractional;; break
   pcase 1?
      echo From ten to nineteen;; break
   pcase *0
      echo Multiple of ten;; break
endswitch
```

This example evaluates the numeric expression `n+m` and compares its string representation against various patterns. The Interpreter simply checks each `case` or `pcase` from first to last until it find one that matches in whichever way is appropriate.

The third way of writing cases is `rcase`, which matches *regular expressions.* For a detail description of regular expressions, see the manual page for *regex(2).* Briefly, a regular expression is like a wildcard pattern but much more expressive. For example, "+" means "one or more of the preceding thing," so "`(foo)+`" would match only `foo`, `foofoo`, `foofoofoo`, etc. You can quote a character in a regular expression by preceding it with a backslash: "\++" means one or more "+" characters. Backslash quoting also works to quote "`*`", "?", and "`[`" in `pcase` commands.

```
switch n
   ⋮
   rcase 9+
      echo Composed only of nines
endswitch
```

The `While` *Loop*

This construct repeats the commands in its body as long as some condition is true. This condition is just an expression, usually employing the `=`, `<`, `<=`, `>`, `>=`, and `<>` operators to compare values, and the `&`

and | operators for "and" and "or." In general, if the expression evaluates to a number it is "true" if the number is nonzero; if it evaluates to a string, it is "true" if the string is nonblank.

```
readnum i : Enter a number:
while i <= 0
   readnum i : Enter a POSITIVE number:
endwhile
```

If the condition is false from the start, the body of the loop is skipped entirely.

Note that if evaluating the expression causes an error, this will abort the View procedure or command file in the usual way. You can treat an erroroneous expression as "false" using the Expander:

```
while ${i <= 0:e:0}
```

If evaluating `i <= 0` gets an error (say, because `i` is undefined), then the default of `0` (= "false") is used instead.

If you omit the condition entirely, the loop runs "forever":

```
while
   readnum i : Enter a number:
   if i > 0
      break
   endif
   echo Enter a positive number, please!
endwhile
```

In this example, we have used a `break` command to break out of the `while` loop prematurely (which in this case means, before the End of Time!).

In a `while` loop, the `break` command jumps directly to the next command after the `endwhile`. Another command, `continue`, jumps directly back to the head of the loop, re-evaluating the condition, if any. Both of these commands are discussed more fully in a later section.

The `Repeat` *Loop*

This loop is much like `while` except that it always executes once, since its condition is written at the end:

```
repeat
   readnum i : Enter a positive number:
until i > 0
```

As before, you can use `break` to break out of the loop, or `continue` to jump directly to the `until`. Also, you can omit the condition on the `until` line to form an infinite loop indistinguishable from an infinite `while` loop.

Note that `repeat–until` is the only built-in control construct whose terminating command is not formed by prepending the word "`end`."

The `For` *Loop*

This construct is especially designed for counting. The syntax is:

```
for i = 0 : 10
   print i
endfor
```

This will print eleven numbers, from zero to 10. The first time through the loop, `i` will equal the starting value `0`. On each successive iteration, it will be increased by one until it exceeds the termination value `10`. If the termination value is below the starting value, the body of the loop will be skipped entirely:

```
for i = 1 : 0
   echo This is never printed!
endfor
```

You can specify a different increment than one by adding a second colon:

```
for i = 0 : 2*pi : 0.1
   print sin(i), cos(i)
endfor
```

Note that `pi` is a built-in constant in the View program. All three values may be arbitrary numeric expressions. If these expressions contain variables which are modified in the body of the loop, their values upon entry to the loop are saved and used.

If the increment is negative, the loop counts downwards. In this case, the loop is skipped if the terminating value is *greater than* the starting value.

The `break` command jumps to the next command after the `endfor`. The `continue` command jumps to the `endfor`, where the loop variable is incremented and tested, as usual.

The `Break` *and* `Continue` *Commands*

These commands modify control flow in looping constructs. The `continue` command acts upon the innermost `while`, `repeat`, or `for` loop. The `break` command acts upon the innermost `while`, `repeat`, or `for` loop, or `switch` construct. The exact action taken depends on the construct, as described in the previous sections.

If you must break out of several constructs at once, you can write an integer after the `break` command:

```
for i = 0:10
   for j = 0:10
      break 2
   endfor
endfor
echo Break will come here
```

The command `break 1` is identical to `break` with no arguments.

Similarly, `continue` with an argument continues the n'th innermost `continue`-able construct.

These commands will generate an error if you specify a larger value than there are applicable active constructs. If you break out to a loop in an outer command file or procedure, the current file or procedure will be aborted cleanly.

The `Try` *Construct: Error Trapping*

This construct is analagous to the `try` statement in HP Pascal. If any errors occur in the body of the `try`, rather than aborting your program they will cause it to jump to the "recovery" clause that you give.

```
try
   readnum x
   print 'The inverse of x is ', 1/x
recover
   echo You naughty user!
endtry
```

If any error that would normally cause a red message to be printed occurs while executing the command between `try` and `recover`, then the Interpreter jump directly to the commands following the word `recover`. If the program reaches the `recover` line without error, it jumps to the `endtry`.

The body of the `recover` clause is *not* protected by the `try` construct. You must use an additional nested `try` to trap errors there.

When the program enters the `recover` clause, the `errormsg` parameter will be set to the error message in string form. For example, this parameter might be "`Division by zero`" in the above example, or "`[STOP]`" if the user pressed the [STOP] key. A second parameter, `errorstr`, is set to the actual error message that would be reported. This is usually a string the same as `errormsg`, followed by a notation that shows the line number and procedure name in which the error occurred.

A program will typically use a `switch` construct to compare `errormsg` against the set of errors it wants to trap:

```
try
   readnum x;; print 1/x
recover
   switch $(errormsg:q)
      case 'Division by zero'
         error $(errormsg)
         echo Sorry, Charlie!
      otherwise
         cerror $(errorstr)
   endswitch
endtry
```

This example introduces two more commands, `error` and `cerror`. The former prints an error message on the screen in red, then goes on. The latter actually "causes" an error with the specified message, but does not add the current line number to the message. The `c` stands for "continue," since this form is usually used to continue an error that was trapped but that could not be handled. A third form, `ferror` (for "fatal" error) generates an error as in `cerror`, but *does* add the current line number. Actually, if done from inside procedure, it reports the line number of the command that called the procedure, rather than of the `ferror` itself.

All three error-message commands take their arguments explicitly. In other words, they are like `echo`, not `print`: if we had wanted to write the error message using `print`, we would have needed to quote it as in

```
print $(errorstr:q)
```

Beware that a loop of the form:

```
while
   try
      ⋮
   recover
      error $(errormsg)
   endtry
endwhile
```

is dangerous, since the [STOP] key is considered an error that is trapped by the `try`. Thus, if there is a bug in your program there may be no way to exit except by trying to press [STOP] twice in a row so that the second [STOP] is received before the `try` has been reestablished. Bad news!

Parameters

Because the values, and even types, of variables are unrestricted, and because programs sometimes use wildcards and expansions to work with variable names, it is bad for the View program to manipulate variables in any way not explicitly requested in the program. Instead, there is a separate set of *parameters* which have more specific uses and behaviors.

To see the values of all the parameters, type `set`. To change a parameter, type:

`set` *parameter* `=` *value*

where the set of things acceptable for *value* depends on the parameter. For example, the `logfile` parameter holds the name of the transcript file, and it expects a new file name for the *value*. On the other hand, `epsilon` is a numeric parameter that wants a numeric expression for *value*.

Here is the complete set of parameters built-in to View:

`debug` This boolean parameter is `True` if the user wants extra debugging information from various user-written commands. Pascal command procedures often examine this parameter to see if they should write out internal information. Few if any of the View built-in commands use `debug`.

`epsilon` This real-valued parameter is the smallest relative difference between two numbers for them still be considered different. Specifically, if a and b are to numbers being compared, then they are considered equal if they are truly equal, or if

$$|a - b| < \epsilon \max(a, b)$$

where ϵ is the value of the `epsilon` parameter. The default value is $\epsilon = 10^{-5}$. If you set $\epsilon = 0$, all tests will be for exact equality.

`errormsg`
`errorstr` These string parameters are normally blank; when an error is trapped by a `try` construct, they are set to the relevant error message. For the latter parameter, the guilty line number and file or procedure name is also included.

`logfile` This is the name of the log file. It is normally `VIEW.LOG`. If you assign a name to it, a default suffix of `.LOG` will be added and the log will be reopened under that name. If you assign it to be blank, no log will be kept. (If you assign it blank inside your `view.cnf` file, View will not create a log file at all.)

`machine` This read-only string parameter contains the type of machine on which the View system is running. This will be "`9836C`" on a color Chipmunk, and "`320`" on a Bobcat (even on Bobcat 330's and 350's!).

`quiet` This boolean parameter suppresses most of the informational messages that View writes from time to time. They will still be written to the log file, if any; they will just be suppressed from the screen.

`seed` This integer parameter is the seed value for the random number generator functions, `rand` and `rand2`. It is initialized to a "random" value based on the real-time clock, and updated every time you use one of these two functions.

`trace` This integer parameter is the "trace level." It is normally zero. If you set it equal to one, View will write each command on the screen before executing it, as a handy debugging aid. If you set it to two, View will write considerably more as it executes your commands.

`user` This read-only string parameter contains the user's login name.

`version` This read-only string parameter contains the version number of this version of View.

`wildcard` This parameter takes on one of the values None, Shell, or Regex. In the default case, Shell, the wildcard characters `*`, `?`, and `[]` are interpreted as patterns in variable names by many commands. For example, `rm a*` will remove all variables whose names begin with `a`. If this parameter equals Regex, then variable names will instead be interpreted as regular expressions. If this parameter equals None, then variable names are taken verbatim. Naturally, in some circumstances wildcards are not supported no matter what the value of this parameter: for example, in an arithmetic expression, `*` always means multiplication!

You can create your own parameters with the `addparam` command.

```
defparam greeting = Hello, world!
```

This creates a string-valued parameter called `greeting`, whose initial value is "`Hello, world!`".

```
defparam [int] max_penguins = 100
```

This creates an integer-valued parameter `max_penguins` initially equal to 100.

In general, the type must be one of:

`[str]` This is the default type, an arbitrary string of characters. In `set` commands the value must follow the `=` sign without quotes.

`[int]` The parameter may hold any integer value. In `set` commands, the value must be a numeric expression.

`[real]` The parameter may hold any numeric value. In `set` commands, the value must be a numeric expression.

`[bool]` The parameter may hold one of the Boolean values `True` or `False`.

`[fn=`*suffix*`]` The parameter may hold any file name. If the *suffix* is specified, then any name entered into this parameter will have that *suffix* added as a default.

If you want to specify further restrictions on the values a parameter may take, you can use `define [before]` to make a special case for the `set` command:

```
defparam [int] max_penguins = 100
define [before] set "max_penguins" = {newval}
   if $newval < 5
      ferror Must be five or more
   else
      builtin set max_penguins = $newval
   endif
enddef
```

If a user tries to give a "`set max_penguins =` *x*" command, the Interpreter will execute the above procedure rather than the built-in definition of `set`. Note the use of the `builtin` command to avoid an unwanted recursion. See the section on *Procedures* for further explanation.

Warning: This method has the flaw that if the user writes an abbreviated parameter name (such as `max_peng`), the above template will pass it through. Currently there is no way around this except to use short parameter names that users will not be tempted to abbreviate!

Input and Output

The simplest way to write to the screen is to use the `echo` command:

```
echo blah, blah, blah.
```

This command simple takes the rest of the line and simply prints it on the screen. It is also written to the log file. This form is rather restricted, though; it must always write a complete line, and can not write a line with leading blanks. More general is `print`:

```
print expr expr expr ...
```

which evaluates and prints each expression in turn. Expressions evaluate to either strings or numbers. If two numbers are written in a row, View adds a blank space between them. Otherwise, the output is exactly what you specify. The expressions may be separated by space or commas. Because of this, you must not write spaces in the middle of an expression except between quotes or parentheses:

```
print 1 + 2    ## bad
print (1 + 2)  ## good
print 1+2      ## good
```

You can use the `[more]` option to suppress the carriage-return at the end, the equivalent of using `write` instead of `writeln` in Pascal:

```
for i = 1 : 10
   print [more] i ' '
endfor
print 'puppies'
```

prints

```
1 2 3 4 5 6 7 8 9 10 puppies
```

It is also possible to write to a file, using the `rewritefile` command:

```
rewritefile f = 'foo.text'
print [file=f] 'Table of Square Roots'
for i = 0 : 10
   print [file=f]  i ' ' sqrt(i)
endfor
closefile f
```

Here `f` is a variable called the *file handle.* The `rewritefile` command assigns to it an integer value that identifies the file for further writes. If you try to give a variable to the `[file=]` option that was not returned by `rewritefile`, dire consequences may ensue.

If you forget to close your file, it will be closed automatically when the View system exits. View keeps only one output file open at a time, so writing alternately to several files may be slow as View constantly closes and reopens the files.

Currently `print` is the only command that supports the `[file=]` option for output files.

To read a string from the keyboard, use `readln`:

```
readln name : What is your name?
```

Here the View program will write the prompt string after the colon and then wait for the user to enter a line. That line is stored directly in variable `name`.

The `[opt]` option specifies that the input is optional. If the user enters a blank line, the variable will retain is old value rather than being set to the null string:

```
readln [opt] name
```

In the above example, View will generate a default prompt of "`name? `".

The `[exp]` option runs the input line through the Expander stage before being assigned to the variable (or possible ignored if `[opt]` is also specified).

```
view> i = 7
view> readln [exp] name
name? number ${i+2}
view> print name
number 9
```

If you write a second word after the variable name, that word is used as the end-of-file marker:

```
readln name -EOF- : What is your quest?
```

Here, if the input stream reaches end-of-file (because the user pressed control-D), `name` will be assigned equal to "`-EOF-`" rather than generating an error.

You can also read from a file:

```
resetfile f = 'bar.text'
while f <> 0
   readln [file=f] buf
   print buf
endwhile
```

The `resetfile` command opens the specified file and returns an input file handle, which is an integer similar to an output file handle. When View executes the `resetfile` command, it actually reads the entire file into memory using the Reader stage. One side-effect of the Reader is that blank lines and comment lines are removed. You must take this into account when planning your file format!

The `readln` command reads one line from the file identified by the specified handle variable, possibly changing the value of the variable to reflect the new state of the input file. When you reach the end of the file, the handle will be zero. Note that if the file contains no non-blank, non-commentary lines, the handle will be zero immediately. Doing a `readln` from a zero handle will generate an error unless you have also specified an end-of-file marker. The Expander is not called for the input lines unless you specify the "`[exp]`" option. Note that "`[opt]`" has no meaning for input files since blank lines are eliminated before they reach the `readln` command. Similarly, it is useless to specify a prompt string.

It is not necessary to close an input file, though it shouldn't hurt to do so.

If you are reading numbers, you can use the `readnum` command:

```
readnum i j k : Enter height, width, and depth:
```

This command reads a line of input, then attempts to interpret it as three numbers (or numeric expressions). The numbers may be separated by spaces and/or commas. If the user does not enter the correct number of values, the command prints a message and asks again. If the correct number of values is finally entered, View assigns these values, in order, to the variables that were listed in the command.

You can use the same `[opt]`, `[exp]`, and `[file=]` options with `readnum` as with `readln`. Note that if the command needs, say, three values and the user enters only two values, then asks for the default when prompted again, *none* of the variables will have been modified.

One last input-related command is `typeahead`, which simply adds some text into the typeahead buffer as if the user had typed it on the keyboard. The line is inserted at the *front* of the typeahead buffer—if you give the pair of commands:

```
typeahead hello\n
typeahead world!\n
readln a;; readln b
```

then `a` will get "`world!`" and `b` will get "`hello`". Note the use of "`\n`" to represent the "newline" (i.e., Enter or Return) key. You can also write "`\e`" for the Escape key, or "`\nnn`", where `nnn` is any decimal number from 0 to 255, to enter the character with that ASCII code. Also, leading and trailing spaces on the line are normally ignored unless "\" appears at the beginning or end of the lien, respectively:

```
typeahead \]␣foo bar␣\
```

where "␣" represents a space, will type ahead "␣␣foo␣bar␣" including the spaces. Naturally, "\\" types ahead an actual backslash character.

The `typeahead` command is most useful when you are writing a procedure to run a command automatically that normally would ask the user for input:

```
use curves     ## plotting and manipulating x-y curves
readln ans : Want a permanent copy of this?
if ans = 'yes' ;; perm = 'A' ;; else ;; perm = '' ;; endif
typeahead My Title\nV\nlog\nA\nlinear\n${ans}\n
plot vgs1 vgs2 vgs3
```

Command Files

The `source` command reads commands from a specified `.view` file. When it reaches the end of the file, it returns to the previous input command stream. The `source` command may be abbreviated, `do`.

```
view> do foo      ## read commands from foo.view
```

The `use` command also checks for `.view` files; `use foo` is nearly synonymous with `do foo`. However, `use` remembers which files have already been used and will not read a given file more than once. `Do` is always happy to re-read a file.

If you want to stop reading from a file before the physical end-of-file you can execute a `quit` or `quit [file]` command. (See the *Miscellaneous Commands* section for details.)

A related command is `dokbd`, which reads commands from the keyboard. When the user presses control-D or enters `quit`, control returns to the caller of the `dokbd` command. This is much like a "shell escape," except that it escapes to interactive View rather than to the Shell. Be aware that if the user gives a command that gets an error, and there is no active `try` construct to trap that error, the `dokbd` command along with the calling procedures and command files will be aborted and the user will be left in the normal, "outer" View shell. To prevent this, you can use the following loop:

```
while
   try
      dokbd
      break
   recover
      error $(errorstr)
   endtry
endwhile
```

Here, if the `dokbd` ever terminates normally (because the user pressed control-D or `quit`), the `break` command will break out of the `while` loop. But, if an error occurs, the usual message is printed and the `while` loop returns to ask for more input commands. This will produce an interactive View shell that more closely resembles the "outer" shell.

Defining Commands

The basic way to define new commands has already been introduced:

```
define foo
   print 'Hi, I''m the FOO command.'
   print 'See how nice I am.'
enddef
```

After this, executing a `foo` command is essentially a shorthand for entering the commands between `define` and `enddef`.

The body of the command is put through the Expander twice. The first time, as the definition is being recorded, only expansions of the form `${{ }}` and `$(( ))` are done. Then, as the commands are actually executed, they are expanded again the usual way. If the body should contain nested `define` commands, still more parentheses may be used to do expansions at the various relevant times.

A command may be given arguments:

```
define multiply a b c
   $a = $b * $c
enddef
```

In this example, if you then did the command

```
view> multiply z 3 7
```

the Interpreter would execute:

```
z = 3 * 7
```

When Interpreter parses the `multiply` command, it sees that `multiply` has three arguments and so tries to read three words from the command line. In this case, it gets "`z`", "`3`", and "`7`", respectively. It then assigns these strings to the variables `a`, `b`, and `c`. Actually, these are *local* variables, which means when the procedure `multiply` finishes their original values will be restored. However, this are not like Pascal local variables which can only be referred to within the procedure itself. In View, when you make a local variable `a` it actually copies the value of `a` to a safe place, assigns to the same old variable `a`, then, when the procedure returns, restores the copied value. So, if the procedure `multiply` were to call another procedure which referred to `a`, that procedure would see `multiply`'s "local" `a`, not the global `a` it might expect. So, it is important to be careful when naming variables.

Since `b` and `c` will in this example get the *strings* "`3`" and "`7`", not the numeric values, you must use `$` format to substitute them into the assignment statement rather than just writing them. In other words, if you had multiplied `b * c` instead of `$b * $c`, it would have failed trying to multiply two strings!

Also, it is a good idea to surround argument expansions with parentheses:

```
   $a = ($b) * ($c)
```

For example, if we wrote

```
view> multiply z x+2 y
```

the computer would have tried to execute

```
z = x+2 * y
```

which would not compute what you expected, since multiplication binds more strongly than addition in expressions.

Also, beware that

```
view> multiply z a b
```

is also doomed to failure because of name conflicts: by the time `$a` is expanded, `a` has the string value "`a`"! So again, it is important to name variables carefully.

If you write the list of argument variables with no punctuation except spaces, then View will divide the calling command line into "words," with one word per argument. A word is a set of characters not including any terminator characters except between quotes or parentheses. A terminator character is a space, comma, `:`, `;`, `=`, `@`, `[`, or `]`. Words are separated by spaces or commas; the other terminators are not allowed at all.

Examples of words are:

```
a+b*c          ## but not  a=b*c
f(a,b)         ## but not  f[a,b]
(a + b)        ## but not  a + b
```

If you include punctuation in the argument "template," you must match the punctuation when you call the procedure:

```
view> define multiply a = b * c
view>> print 'a=' a ', b=' b ', c=' c
view>> enddef
view> multiply z = 2*3
a=z, b=2, c=3
```

If an argument variable is followed by a punctuation symbol, that symbol becomes a new terminator even if it would not ordinarily be. For example, the argument `b` would normally match against the whole word `2*3`, but since that argument is terminated by "`*`" it stops as soon as it sees a `*` in the input command. However, the thing that matches with `b` must still be a valid word, that is, it should not contain any other terminator characters, and it should not be blank. Any leading or trailing spaces around the word are ignored. For example:

```
view> multiply z = 3+4*7      ## works, but beware: it computes (3+4)*(7)
view> multiply z = 3 * 7      ## works
view> multiply z = * 7        ## fails, because b is blank
view> multiply z = 3 *        ## fails, because c is blank
view> multiply z = (a * 3)*6  ## works, with b='(a * 3)' and c='6'
```

Any character in the argument template that doesn't belong in a variable name is interpreted as a punctuation character. If several punctuation characters appear next to each other with no spaces between, they form one symbol which must be matched exactly. If punctuation characters are separated by spaces, they may be matched with or without spaces:

```
view> define foo a += b ;; enddef       ## first definition
view> define foo a + = b ;; enddef      ## second definition
view> foo yow += 17+3         ## matches first and second defs
view> foo yow + = 17+3        ## matches only second def
```

Any word between quotes in the template is also matched exactly:

```
view> define count min "to" max ;; ...
view> count 17 to 34          ## works
view> count 17 TO 34          ## works: templates are case-insensitive
view> count atob              ## fails: "to" must not be part of a name
view> count 17to34            ## fails
view> count (17)to-4          ## works, since ) and - are not name chars
```

However, it is never possible to require blanks:

```
view> define foo1 a + = b ;; ...
view> define foo2 a '+ =' b ;; ...
```

The above definitions have exactly the same effect.

It is also possible to create argument variables that are not restricted to matching words:

```
view> define foo first second {rest}
view>> print 'first=' first ', second=' second  \\
view>>>      'rest=' rest
view>> enddef
view> foo a b c d e
first=a
second=b
rest=c d e
view> foo a, b, c, d, e
first=a, second=b, rest=c, d, e
view> foo a b:c
first=a, second=b, third=:c
view> foo a b
first=a, second=b, third=
```

Basically, an argument variable surrounded by curly braces in the template, like `rest` in this example, may match any number of characters with no restrictions except that quotes and parentheses must be balanced.

```
view> define multiply v = {a} * {b};; ...
view> multiply z = 3 * 4      ## a = '3', b = '4'
view> multiply z = 3*4*5      ## a = '3', b = '4*5'
view> multiply z = (3*4)*5    ## a = '(3*4)', b = '5'
view> multiply z = *          ## a = '', b = ''
```

It is illegal to have an argument variable in braces followed by another argument variable with no intervening punctuation, as in

```
view> define foo {a} {b}      ## WRONG
```

because `b` could never match anything that wouldn't be matched first by `a`.

You may ask, *why do we care* about these punctuation rules, anyway? Why require the user to type punctuation marks that are essentially meaningless? The answer lies in View's ability to record several definitions for a command.

```
define count min : max
   for i = $min : $max
      print i
   endfor
enddef

define [after] count max
   count 1 : $max
enddef
```

Here, we have given two definitions for the command `count`. Note that the `[after]` option has been used on the second definition. It says, rather than replacing the definition of `count`, I want to *augment* the existing by adding a new definition after the current one(s). Now, when View fails to match the arguments against the template for the first definition, it can go on to try the second template instead. It will fail only if *none* of the templates match. In the above case, if you write

```
view> count 3:7
```

the first template matches and so View never uses the second definition. But, if you write instead

```
view> count 8
```

the first template fails but the second succeeds. So, the second definition is executed resulting in

```
count 1 : 8
```

This may seem like it's not getting anywhere, since we have swapped one `count` command for another, but this new `count` command will now match the first template. In effect, we have just define a command with optional arguments. And, rather than having to write out all the cases in full, we can just define all the more specific cases in terms of the general case.

Taking another example,

```
# Step 1.  Parse variable name, default of i
define count v = {rest};; _count $v = $rest;; enddef
define [after] count {rest};; _count i = $rest;; enddef

# Step 2.  Parse optional increment, default of 1
define _count v = min : max
   _count $v = $min : $max : 1
enddef
define [after] _count v = min : max : incr
   for $v = $min : $max : $incr;; print $v;; endfor
enddef
```

Here it was easier to define an intermediate command, `_count`, which can assume it has a valid variable name `v`. The `count` command is just responsible for filling in a default variable name.

Any command name beginning with an underscore "_" is "invisible." That is, if the user types `??` to get a list of all commands, invisible commands won't be included in the list. That seems appropriate here, since users are never supposed to run `_count` directly.

For example, if the user enters

```
view> count 2:4
```

then this will match the second template for `count` and so the Interpreter to proceed to execute

```
_count i = 2:4
```

This will match the first template for `_count`, which will turn into the command

```
_count i = 2:4 : 1
```

which is then executed as a `for` loop.

As still another example, you can parse a number of arguments using recursion:

```
define count num {rest}
   for i = 1 : $num;; print i;; endfor
   count $rest
enddef

define count
   ## Just do nothing at all
enddef
```

Here, if the user types

```
view> count 3 5 2
```

this will match the first template with `num` equal to "3" and `rest` equal to "`5 2`". The procedure will then count to 3, then call itself:

```
count 5 2
```

which will count to 5 and call

```
count 2
```

will will count to 2 and at last call

```
count
```

which will return trivially. This is good ol' fashion recursive divide-and-conquer programming. View optimizes tail recursion, so the above example is not only easy to write, but efficient, too! (Tail recursion optimization means that since `count $rest` is the last command in `count`, View treats it more like a "goto" than a procedure call. So, you can follow an unlimited number of tail recursion steps without getting a "stack overflow" error.)

You can return prematurely from a procedure with the `return` command.

```
define yes_or_no  v default
   for i = 1 : 3
      readln $v : Yes or no?
      switch [ci] $v
         pcase y*;; pcase n*
            return
      endswitch
   endfor
   print 'I will assume you mean ', default
   $v = default
enddef
```

If the user enters a line beginning with "y" or "n," the procedure will return. If the user still can't get it right after three tries, the procedure assumes they mean no and returns the normal way.

No command is complete without help information, so there is an `addhelp` command for adding to the help message that is printed when the user types "`?`*command*".

```
define count min : max ;; ...
define count max ;; ...
addhelp  count <min>:<max>
addhelp >  Count from <min> to <max> and print each value.  Default
addhelp >  for <min> is 1.
```

Note the > symbols for indenting lines. If the argument to `addhelp` begins with a > symbol, View deletes that symbol but retains any spaces that follow it.

Some commands want several names. For example, the `copy` command is also available as `cp`, just for Unix fans. To establish an alternate name for a command, use `alias`:

```
procedure count min : max ;; ...
alias step = count
```

You can also alias the built-in commands, except for control-construct commands.

Another handy command for working with procedures is `undefine`. The command

```
view> undefine count _count
```

will remove all current definitions of `count` and `_count`, if any, including built-in Pascal definitions. The `define` command with no `[before]` or `[after]` option does this automatically, too.

Finally, `editdef` lets you edit the definition of a command:

```
view> editdef count _count
```

will write all current View language definitions of `count` and `_count` to a temporary file, then call up the Caged editor on that file. When you push the Caged "exit" key ("(" on the keypad), View will re-read the definitions into its memory. Of course, a more permanent way to edit commands is to keep them in a permanent file, then edit that file with a shell-escape.

Local variables

In the last section we introduced the concept of *local variables.* You can also define your own local variables aside from the arguments, using the `local` command:

```
define count min : max
   local i
   for i = $min : $max
      print i
   endfor
enddef
```

Now, the `for` loop inside the procedure will not affect the variable i used in the rest of the program.

As mentioned before, variables in View are dynamically scoped and so if the procedure `count` called another procedure that referred to `i`, that procedure would see the local version of `i`, not the global `i` that has temporarily been hidden. This means you must be more careful programming in View than in, say, Pascal, but it also gives a number of advantages to the clever programmer.

You can write a whole list of variable names after the word `local` if you wish, separated by spaces or commas. The new local variables start with value 0.0 unless you explicitly initialize them:

```
define count ...
   local i = 17;; local j = i * 2
   ⋮
```

When you are initializing variables you must write just one variable per `local` command.

It is also possible to declare local parameters:

```
define count
   localparam trace;; set trace=2
   for i = 0 : 10
      print i
   endfor
enddef
```

In this definition, the `trace` parameter is set to two for the duration of the command. When `count` finishes, `trace` is automatically reset to zero or whatever other value it had originally. Notice that the names you give in `localparam` must be for *existing* parameters; if you want to make a new parameter you must use `defparam`, which is typically not used inside a procedure.

As with variables, you can initialize local parameters, effectively combining the `localparam` and `set` commands:

```
define count
   localparam trace = 2
   ⋮
```

Local parameters, unlike variables, keep their previous values. In otherwords, the above `localparam` statement saves away the current value of `trace` but leaves the parameter the same. This helps guarantee that every parameter always has a legal value.

Still more obscure, you can define local commands:

```
define count
   localdef write
   alias write = print
   for i = 0:10 ;; write i ;; endfor
enddef
```

While this may seem silly, it actually comes in quite useful for block procedures, discussed in the next section. As for local variables (and unlike local parameters), `localdef` saves the current definition of the command and then undefines the command until further notice. So there is, in a sense, a "conservation of definition" in effect. View can't afford to copy entire definitions of commands, so it just shuffles them around!

Again, `localdef` + `alias` can be written as an initialization:

```
define count
   localdef write = print
   ⋮
```

Block Procedures

A *block procedure* looks like a control construct rather than a single command. For example, suppose you wanted to write a `forargs` loop that executed a block of code for every argument. (When a `.view` script is started up by the Shell, the number of command line arguments is stored in variable `argc` and the arguments themselves are in `arg1` through `arg$argc`.)

```
define [block] forargs forargs_v
   local forargs_i
   for forargs_i = 1 : argc
      $forargs_v = arg$forargs_i
      forargsbody
   endfor
enddef

forargs name
   if name <> 'Joe'
      print 'hello, ' name
   endif
endforargs
```

In this case, when the Interpreter scans the `forargs` command it makes a local variable called `forargs_v` as usual and assigns "`name`" to it, but it also reads the lines following the `forargs` line, up to the next matching `endforargs` command, and defines that as a local command called `forargsbody`. It then executes the definition of `forargs` as usual; `forargs` simply executes `forargsbody` whenever it wants to execute the body of its loop. When the `forargs` returns, the caller resumes at the next line *after* the matching `endforargs`. The `endforargs` itself is never executed.

We define the `forargs` procedure to use `forargs_i` and `forargs_v` rather than just `i` and `v` just to minimize the danger of name conflicts. It is always a good idea to choose names like these when one command's code may affect another's.

Since `forargsbody` is a *local* command, it is okay to have nested `forargs` commands. At each nesting, a new local `forargsbody` command is created and then undefined automatically as the inner `forargs` returns. The Interpreter counts the nesting of `forargs` and `endforargs` lines as it reads in `forargsbody`, so you don't have to worry about it. However, as with the built-in constructs the scanner does not understand abbreviations or aliases.

You can also define your own name for the ending line:

```
define [block=endfargs] forargs ...

forargs
   ⋮
endfargs
```

It is best, though, to use the default notation unless there is a good reason not to (as in the case of `repeat` and `until`).

Things you can not currently do that built-in constructs can include executing only part of the body (as the `if` statement does), allowing arguments on the closing line (as `until` does), and providing complete control over `break` and `continue` within the body. But you can come close by defining commands:

```
define [block] await_word awaited_word
   localdef check
   define check checked_word
      if $checked_word = ${{awaited_word}}
         ferror 'found awaited word'
      endif
   enddef ##check
   try
      while
         await_wordbody
      endwhile
   recover
      if $(errormsg:q) <> 'found awaited word'
         cerror $(errorstr)
      endif
   endtry
enddef

await_word 'please'
   readln buf : What's the magic word?
   check buf
   echo No, the other magic word.
endawait_word
echo Very good!
```

In this more lengthy example, we define an `await_word` construct that repeatedly executes a sequence of commands which should include a `check` command. When the argument of `check` is the awaited word or string, we break out of the loop. One way to break the loop is to use `break`; another, shown here, uses a `try` construct and generates a fake error. `Try` opens up the possibility of having multiple exits from the loop.

The definition of `check` uses two braces so that `awaited_word` is expanded as `check` is being *defined.* One or no braces would expand as `check` is *executing*; actually, this would have worked just as well. Three braces would expand as `await_word` itself is being defined.

Miscellaneous Commands

The `quit` command exits from one level of command-stream input. In other words, if the command appears in a procedure, it acts like a `return` command. If it appears in a command file, it acts like a premature end-of-file. If it is typed from the keyboard, it exits View (or the current `dokbd` command, if any). You can also use options to modify the behavior of `quit`:

```
quit [file]       ## exit from topmost command file
quit [proc]       ## exit from topmost procedure
quit [all]        ## exit all the way out to outer-level View shell
quit [view]       ## exit from the View system
```

For example, if command file `foo.view` calls procedure `bar`, which does a `dokbd` command, and the user enters `quit [file]`, then all the input streams back to and including the file `foo.view` will be aborted. View will read the next command after the one that invoked `foo.view`. The word `exit` is an alias for `quit`.

The `bye` command is equivalent to `quit [view]`. The `return` command is equivalent to `quit [proc]`. Getting an error that is not trapped by a `try` construct is similar to `quit [all]`.

The "`!`" command is a shell-escape. If used by itself, it starts an interactive Shell. If followed by a Shell command, it executes that command and returns. If the command gets an error, the Shell will print a message and then the "`!`" command will fail with `errorstr` and `errormsg` both blank. Although shell-escapes are most often done from the keyboard, they work in procedures and command files, too:

```
define print {curves}
   save /tmp/t$(user) {curves}
   try
      ! print -pa /tmp/t$(user)
   recover
      ferror Unable to print /tmp/t$(user)
   endtry
enddef
```

The above command saves the curves in memory to a temporary file (whose name is derived from the user's name), then prints that file on the Apple laser printer.

The "`?`" command gives Help information about various subjects. It can also be written `help` or `man`. By itself, it displays a list of common View commands with brief descriptions, plus a list of additional commands that users might find helpful. If the "`?`" is followed by a command name, it displays whatever help information is available for that command. You can use `addhelp` to record this information about your own procedures. If you define a procedure and don't use `addhelp` for it, then "`?`" will simply report the kinds of arguments the procedure expects. You can also follow "`?`" with a parameter name to see what kind of data that parameter expects.

The "`??`" command prints a complete list of commands. All commands and aliases whose names do not begin with "`_`" are included. Initial characters of the command names are highlighted to show the shortest legal abbreviation for each command. If you give a command name, it writes out the current definition(s) for that command.

The `history` command shows the keyboard commands you have entered recently, in chronological order. Given a numeric argument, it lists that many commands.

The `interp` command displays a list of currently defined interpolators. These are described in the *viewmod(3)* documentation.

Examples

Here is a procedure for reversing a curve, i.e., putting the x- and y-values in front-to-back rather than back-to-front order.

```
define reverse curve
   o$curve = $curve              ## Save old curve
   reverse $curve = $curve       ## Reverse 'in place'
enddef

define reverse new = curve
   local i len xc yc             ## Local variables

   yc = $curve                   ## Create temporary curves to poke into
   xc = ${curve}_x               ## _x preserves units on the basis
   len = cvlen(${curve:q})       ## Number of data points
   for i = 1 : len               ## Copy x- and y-values backwards
      poke yc @ i = cvidx(${curve:q},      len-i+1)
      poke xc @ i = cvidx(${curve:q}+"_x", len-i+1)
   endfor
   splice $new = xc : yc         ## Create final curve
enddef
```

Here is a procedure, borrowed from a CNS 185 assignment, for setting up a system of differential equations for use by the `solver` tool. This tool defines an `eqn` command for recording the equations to be solved:

$$V'_{ij} = I_{ij} - f(\sum_k V_{ik} - 1) - f(\sum_k V_{kj} - 1) - f(-V_{ij})$$

where

$$f(x) = \alpha e^{\alpha x}$$

We simply create N^2 View variables, `V11` through `V`NN, to hold the results. (If N were greater than 9 we would have to use a form more like `V1_1` or `V1#1`.)

```
define setup1
   local i j sumik sumkj
   eqn [clear]    ## Clear existing equation system
   for i = 1:N
      for j = 1:N
         echo Setting up V$i${j}...
         sumik = '-1'
         sumkj = '-1'
         for k = 1:N
            sumik = sumik + "+V$i$k"
            sumkj = sumkj + "+V$k$j"
         endfor  ## loop on k
         eqn [and] V$i$j` = I$i$j - a*exp(a*($sumik)) \
                                  - a*exp(a*($sumkj)) \
                                  + a*exp(-a*V$i$j)

      endfor  ## loop on j
   endfor  ## loop on i
enddef
```