TABULA VIGILANS

User Manual – Version 1.5

by Richard Orton

HTML version ed. A. Endrich

Acknowledgements

There are many who have assisted in the design of *Tabula Vigilans*, but I should like to single out for special mention Martin Atkins, who helped considerably with a complete re-design of the parser using the Yacc utility. He and many others helped by simply being available to discuss the design of rules: I should like to mention especially Archer Endrich, of the Composers' Desktop Project, and my colleagues at the University of York, Ross Kirk, Andy Hunt, Tony Myatt, David Malham, Nick Fells and Mark Pearson.

Richard Orton,
York, 1994 – 2000; Willoughby on the Wolds, 2007

CONTENTS – INTRODUCING THE ENVIRONMENT

CONTENTS – LIST OF KEYWORDS

List of Keywords ctd.

int2string	Type conversion	converts an integer value to a string
interp_table	Rule	interpolate a value between each corresponding value in two tables
lim	Rule	constrain a cell to lie within given limits
lin	Rule	create a linear time-varying output
lintrans	Rule	multiply and add combined in a single (linear) operation
local	Control-flow	declare a cell to be local (private) to the current procedure
lock	Rule	lock one or more cells
log	Rule	create a logarithmic time-varying ouput
log10	Rule	create a logarithmic time-varying ouput
loop	Control-flow	return control-flow to to the start of the current procedure
max	Rule	output the maximum value of a number of input arguments
mean	Rule	output the average, or mean, of a number of input arguments
message	Rule	output a message string to the console
messag1	Rule	output a message string to the console only the first time it is called
midichord	Rule	output a chord to the designated MIDI channel
midiecho	Rule	output immediately (echo) a MIDI event
midiin	Rule	collect and store input MIDI data
midiout	Rule	output a MIDI message
midiset	Rule	set instruments to specific MIDI channels
min	Rule	output the minimum value of up to nine input cells
mouse	Rule	specify mouse position
mult	Rule	multiply the values of any number of input cells
mult_table	Rule	multiply the values of a table by those of another table
num2string	Type conversion	converts a floating-point value to a string
offset_table	Rule	offset the contents of a table by a value
perm	Rule	randomly permutate the contents of the input table
pitchbend	Rule	send a MIDI pitchbend message
pop	Rule	fractal algorithm which employs an output cell and an input value
power	Mathematical	power function
print	Rule	print an input argument to the console
probe	Rule	display the input arguments to 2 decimal places on the console
probi	Rule	display values as integers on the console
rand	Mathematical	generate a random number between 0 and 1
random	Mathematical	generate a random number between two specified values
return	Control-flow	return to a previous procedure
round	Mathematical	return the nearest rounded integer

List of Keywords ctd.

scale_table	Rule	scale each cell in a table by a specified constant
schedule	Rule	schedule a MIDI event for later performance
seg	Rule	create a linear time-varying output
shift	Rule	shift contents of a table one place to the left (or right)
showargs	Rule	show commandline string arguments (or right)
sine	Mathematical	give the sine of a value in radians
sort	Rule	sort a table into ascending or descending order
sqrt	Mathematical	give the square root of a value
store_digits	Rule	set number of decimal places to store values in a file
store	Rule	store values to 2 decimal places in a file
storf	Rule	specify filename(s) for additional store functions
stori	Rule	store values as integers in a file
storefile	Rule	specify filename(s) for (multiple) store functions Also see **storf**
storestr	Rule	store an ASCII string in a file
subst	Rule	draw values out of two input tables with statistical weighting
sum	Rule	sum in the output cell the input arguments
sum_table	Rule	sum successive values in two tables
swap	Rule	swap the values of two cells
switchon	Control-flow	alternate conditional branch construct
system	Operating system call	passes control to a named external program
table	Initialisation	create a table of given dimensions and sizes
tangent	Mathematical	the tangent of x
time	Rule	set a real-time counter
trigger	Rule	set a trigger cell
try	Mathematical	ascertain the return status of a rule
unlock	Rule	unlock previously locked input cells
wait	Rule	cause a complete stay of execution for the duration of the input cell
while	Control-flow	loop dependent on the evaluation of a test
xad	Rule	extract adjacent differences
xar	Rule	extract adjacent ratios

Introduction to *Tabula Vigilans*
Getting Started – Basic Terms & Concepts

Tabula Vigilans is a rule-based algorithmic composition system. The user writes a *script* – a text file – which contains the set of rules defining the composition. The file is submitted to the program which then sets up appropriate data structures for the performance of the composition. The performance itself consists of one or more classes of event:

- Sound Output via MIDI synthesiser(s)
- Messages to the computer console
- Data written to a file
- Graphical output to a screen (not incorporated in this version)

The performance may be directed or at least partly determined by one or more of a combination of input events:

- File and command line input
- Tracking position and movement of the mouse
- MIDI Input

This Manual gives the format of the script, and lists the set of rules which the composer may incorporate in it.

Section 1.0 Introduction

Computer systems appropriate to the version of *Tabula Vigilans* referred to in this Manual are any of those in the PC and SGI ranges. For the full use of MIDI and sound capabilities, additional hardware may. sometimes be necessary. A multi-timbral MIDI synthesizer is recommended for sound output. The author has used the Roland *Sound Canvas* SC-155 to test and develop the system, but example scripts may be edited for use with any multi-timbral synthesizer.

Section 1.01 How to use the program

Assuming the program is in the current directory, the MIDI synthesizer is connected to the computer via MIDI leads, and the audio connections from the synthesizer are connected to a suitable amplifier and loudspeakers, **then**:

- There should be a script in the current directory – either one supplied as an example script with the *Tabula Vigilans* release or one made by the user with a text file editor from the rule definitions supplied with this Manual. In due course, when many scripts are being used, it may be convenient to have a separate directory for the scripts and data files, with a path through to the *Tabula Vigilans* program. In this case the current working directory will be that containing the scripts.
- The overall command line looks like this: `tv [-I -iN -oN -v] rule_file`
- The script (rule_file) should have a name with a ".tv" extension. Let us assume for our immediate purpose that its name is "turtle.tv". Then, from a commandline environment (such as DOS or COMMAND.TOS) the program can be initiated by typing:

  `tv turtle.tv <RETURN>`

  - The program parses the script, sets up all the data structures required, and sends a message to the console indicating that the performance is ready to begin. The performance can then be started by pressing the <RETURN> key. The performance will begin. If there is no 'natural' end to the script, or if the user wishes to terminate the performance early, then the <RETURN> key (Atari) or <CONTROL-C> (PC and SGI) should be pressed again; the performance will end with a message saying how long the performance took.
  - If it is wished to start the performance as soon as possible, without referring back to the user with a 'Ready to run performance' message, then the **-I** flag (for 'instantaneous') can be used, as in:

  `tv -I turtle.tv <RETURN>`
- **Other flags:**

  - **-i***N* – use MIDI IN device *N* (*N* > 0). 0 is the default, using the internal synthesizer.
  - **-o***N* – use MIDI OUT device *N* (*N* > 0). 0 is the default, using the internal synthesizer.
  - **-v** – verbose mode

- If there is a syntax error in the script, then an error message will be reported to the screen; the user should then check the syntax in the appropriate place in the script with the help of this manual. Sometimes errors in the script will not be found until the script is being performed. Again, an error message will be reported on the console, and the user will need to correct the script before trying a re-run.

Section 1.1 Some terms

In *Tabula Vigilans*:

- The **script** is a text file that is submitted to *Tabula Vigilans* for performance. A script comprises one or more **procedures**, of which one must be named *start()*. Other procedures may be named at will by the user.
- A procedure consists of the **procedure_name** followed either by empty parentheses – ( ) – or parentheses containing the name(s) of a cell whose value is being inherited from the calling procedure, followed by any number of lines of script enclosed in brace delimiters: '{' at the beginning of the procedure, and '}' to terminate the procedure.
- A **cell** is a storage location in computer memory. A cell may contain effectively any real number.
- Groups of cells are called **tables**. Tables can be of different dimensions, and they can be of any **size** in any **dimension**.
- Lines of script within procedures may be of three types:
    1. **assignments** – assigning values to cells
    2. **rules** – operations upon cells and tables
    3. **control-flow operations** – determined by the keywords:

        break, **call**, **continue**, **for**, **if**, **if-else**, **loop**, **return**, **while**

- A **primitive rule** (hereafter called simply 'rule') forms a single line of text in the script. The action that *Tabula Vigilans* performs depends on the keyword for that primitive rule, called an **opname** (operation name). Most opnames have input or output cells or tables. Outputs appear to the left of the opname, inputs appear to the right.
- Each rule has a formal **template** as given for each opname in this Manual. The template indicates how many inputs or outputs the rule is expecting, and whether they are cells, tables, or numerical values.
- The completed script (the **rule-file**) forms the **rule-set** for that composition.
- Some operations need to occur before the performance of the script. These are called **initialisation** rules and are defined at the head of the script, outside any procedure.

Section 1.2 Format of the *Tabula Vigilans* script

The broad format of the rule-file is as follows:

```
[ initialisation rules, if required ]
start() procedure
```

[other procedures as required]

Section 1.3 Initialisation Rules [optional]

1.31 #include

It is possible to split a Tabula Vigilans script into two or more files using the **#include** statement. As an example, I have a large script called 'scorebld.tv', over 2000 lines of code in length. In order to be able to work with a smaller file – the one where the 'current development' is, I am able to split the script into two: one called 'sb1.tv' and the current one (in development) called 'sb2.tv'.

At the top of the 'sb2.tv' file I put the statement:

```
#include "sb1.tv"
```

and by invoking the script as

```
tv sb2
```

everything works as if it were one file.

Caution: Remember that error messages with line numbers will only refer to the file with the error in – so you will have to work out which file the error is likely to be in. I have not, thus far, used more than two **#include** files, but in principle any number of files could be included.

1.32 Table declaration with size

Table requires initialisation at the head of the script – i.e., outside any procedure. The user declares the name, dimensions and size of the table(s) required.

Examples of table declarations are as follows:

```
table PENT[20]
```

Here 'table' is the opname, i.e., the *Tabula Vigilans* keyword. The name of the table, chosen by the user, is PENT (here so named because it is to hold a set of PENTatonic piches) – and there are going to be 20 values, assembled in a single-dimensioned array. There is a formal requirement that the first letter of a table name be upper case. By convention, the entire table name is given in upper case, in order for the user to distinguish easily table names from cell names, which are always in lower case.

```
table PURCELL[60][3]
```

Here the name is PURCELL, and it is a two-dimensional table. (By way of example, this might be intended to hold 60 notes, each with an associated duration and dynamic.)

NB To ascertain the number of dimensions a table has, count the pairs of square brackets [ ] after the name. The *size* of each dimension is the number inside the squared brackets.

1.33 Table declaration without size

As long as tables are filled from an external file using **fill_table** it is possible to declare them without specifying the size – the compiler will create a table of the correct size automatically. This makes it easier to use longer tables such as those that might be generated by another program or TV script.

For example, the following will work correctly:

[At the top of the file]
```
table PTABLE[]
```

[and subsequently – probably at somewhere near the beginning of **start()**]
```
PTABLE fill_table "pitches.txt"
```

and "pitches.txt" may contain any number of (e.g.) MIDI note-numbers, as:

```
70 73 76 45 84 60 62 ...[etc.]
```

1.34 String tables

From version 1.5, TV supports string tables. These are identified by prefixing the character ' to the table name. String tables may be initialised with or without size (the latter being followed by **fill_table**). Multi-dimension string tables are not supported. String variables are also currently not possible:

```
astring = 'STRTABLE[0]
```

will produce a syntax error.

Section 1.4 Procedure format

All procedures have the same general format: the procedure-name followed by parentheses, and any number of rule-lines enclosed by opening and closing braces. One of the procedures must be named *start()*, which is where the performance is to begin:

```
start()
{
    [ any number of rule-lines, one to a line ]
    [ loop instruction, if required ]
    }
```

The following is an example of a complete, though short, script, with one procedure only:

```
start()
{

    // set MIDI channel 0 to Instrument 0
    midiset 0, 0
    // read mouse location, and scale output values
    x, y mouse 48, 84, 30, 100
    // display x and y output values messag1 "Pitch\tAmplitude\n"
    probi x, y
    // send a MIDI output with x as pitch, y as velocity
    // and a randomly-derived duration
    midiout 0, x, y, 0.2 * int(random(1, 4))

loop
}
```

Other procedures follow the same broad format, but may be named freely excluding only *Tabula Vigilans* keywords. Each is indicated by parentheses **( )** after the procedure-name, in a format similar to that of the **start( )** procedure, and enclosed in braces, e.g.:

```
watery()
{
```

Again, within the procedure, the keyword 'loop' can be employed to send control back to the beginning of the procedure (in this case, to the beginning of 'watery').

Section 1.5 Comments

A user comment may be placed in a script at any point after a pair of forward slashes (//). These have the special meaning for *Tabula Vigilans* that the rest of the line is ignored. Such comments may also be used in any data files (see Manual entry for **fill_table**) created for performance.

Section 2.0 Rule-Lines

Rule-lines in the *Tabula Vigilans* script may be one of three types: assignments, rules, or control-flow indicators.

Section 2.1 Assignments

An assignment takes the form cell_name = expression, e.g.

```
a = 0.25
```

where a cell (whose name here is 'a') is explicitly assigned a numerical value. The following assignment

```
a = b
```

means that the value of cell 'a' is given the same numerical value as that of cell 'b'.

Section 2.11 Mathematical Expressions in assignments

More complex expressions may be evaluated in assignments and as input arguments to rules and call, etc.. For example,

```
(b * c)/d
```

is a valid expression. The mathematical operators that can be used in assignments are:

+	-	*	/
%	^	&&	||
plus	minus	multiply	divide
modulo	power	logical AND	logical OR

Parentheses may be used to group items in an expression. For example,

```
b^c/d
```

is different from

```
b^(c/d)
```

In such cases, parentheses may be used to force the operators into the associations the user intends.

Section 2.12 Expression shorthand

Some expression shorthand has been implemented, adopted from 'C'. These are given below, with their 'longhand' form: the forms are identical in meaning:

Expression Shorthand & Longhand

Shorthand	Longhand
a = ++b	b = b + 1 a = b
a = b++	a = b b = b + 1
a = --b	b = b - 1 a = b
a = b--	a = b b = b - 1
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

Section 2.13 Mathematical Functions

There is a group of mathematical functions that can be used within assignments or as input arguments to rules. These are listed below, together with the full name of the value they return:

Table of Mathematical Functions Available in *Tabula Vigilans*

abs(x)	absolute value of x
arccosine(x)	the arccosine of x
arcsine(x)	the arcsine of x
arctangent(x)	the arctangent of x
arg(n)	the numerical value of the nth commandline argument
args(n)	the string value of the nth commandline argument
cosine(x)	the cosine of x
dimensions(TABLE)	the number of dimensions a table has
dimsize(TABLE, dimension_number)	the size of the dimension indicated
log10(x)	the base 10 logarithm of x
int(x)	the integer value of x
natlog(x)	the natural logarithm of x
power(x, n)	x to the nth power
rand()	a random number between 0 and 1
random(x, y)	a random number between x and y
round(x)	x rounded to the nearest integer
sine(x)	the sine of x
sqrt(x)	the square root of x
tangent(x)	the tangent of x
try(rule)	the status of executing the rule

All of these mathematical functions can be used in the following form:

```
a = sqrt(x)
```

and also in more complex expressions such as

```
1 + sqrt(x^(b/7))
```

try can be used in the same places as mathematical functions – see Section 2.3 below.

The Manual Pages below contain full descriptions of both mathematical functions and rules.

Section 2.2 Rules

A **rule** consists of a single line of text, comprising (from left to right) output (cell or table) name(s), opname, and input (cell or table) name(s). Valid separators between cell/table names and the opname are spaces or tabs, while commas may be used to separate multiple inputs or outputs. Each primitive rule associated with an opname has a formal template as given in the manual pages, which indicates how many inputs or outputs a rule should have. For example, some rules have no outputs, some have no inputs. Some have fixed numbers of outputs or inputs, while others permit a variable number of them.

Cells may be named freely by the user, the only restriction being that the first character of the name should be a lower case letter. There is no restriction on the length of the name, though spaces within the name are not allowed: the underline character (_) may be used to separate significant name-syllables. Numerical values (such as 3 or -5.7) may also be used as constants in place of cells; these may be integer or floating point numbers as needed.

Section 2.3 try - testing the return status of a rule

The keyword **try** can be used in conjunction with a rule to ascertain the return status of that rule. An example where this is very useful is with the rule **midiout**, which outputs a MIDI message if it is not already in the process of doing so. The return status of midiout is greater than zero if it has succeeded in sending out one or more MIDI note-on messages, and less than zero if the matching note-off has not yet been sent. The use of **try** will mean that updating of channel, note, velocity and duration parameters can take place before the next note or notes are due to be started.

The appropriate rule-line would be in the following form:

```
if(try (midiout chan, note, amp, dur, 1) > 0) {
     ....[update chan, note, amp, dur, etc]
```

Section 3.0 Control flow within a script

Performance of the script begins at the **start()** procedure. If start()is the only procedure, and if there is no loop instruction, then there is no repeated execution, and the performance ends having traversed the start() procedure just once! If there is a loop instruction, then control will be passed back to the beginning of the start() procedure and the cycle of rules will be iterated repeatedly. There are, of course, methods for ending a performance from within the script - for example, by testing some varying cell value. But it is possible for some performance never to have a formal method of ending; in these circumstances the user simply terminates the performance by pressing the <RETURN> key (Atari) or <CONTROL-C> (PC and SGI) on the computer console.

Section 3.1 The *while()* loop

Another way of creating an infinitely repeating loop is to employ a **while()** loop. The extent of a **while()** loop is delimited by curly braces, as in:

```
while(x) {
     ....a number of rule-lines
     ....a number of rule-lines....
}
```

This rule tests for the condition within the parentheses, and if the value tested (e.g. 'x') is non-zero, then the while loop is iterated. So if the user places a constant within the while loop, as in

```
while(1) {
     ....
     ....
}
```

then clearly the value of '1' can never change, and so the performance will continue indefinitely.

It is common for conditional tests to be set within the while parentheses – see Section 3.3, Conditional Evaluations, below.

Section 3.2 The *for()* loop

A method of controlling a definite number of executions of a loop is provided by the **for()** mechanism. Here a counter is set up, which determines the number of times the loop is performed. The form of the for() loop is as follows:

```
for(i = 0; i < 10; i += 1) {
     ....
     ....
}
```

Within the **for()** parentheses are three subdivisions, separated by a semicolon. The first of these assigns the initial value to a counter, the second contains a conditional test which, if evaluated as TRUE, determines that the following loop of rules will be executed once more, and the third indicates what action is to be taken after each iteration.

[Caveat,'C' programmers: do not use 'i++' in the last division of for(); the formal definition of *Tabula Vigilans* demands that this be an assignment!]

In the case shown above, 'i' is initially set to 0, the loop is performed as long as the value of 'i' has not reached 10, and its value is incremented by 1 after each iteration. So in this case the loop will be iterated 10 times. Of course, the following is also legal in *Tabula Vigilans*:

```
for(pitch = highC; pitch > lowD; pitch = pitch - interval) {
     ....
     ....
}
```

The structure of the *for()* loop is the same, but here 'highC', 'lowD' and 'interval' are cells whose values will have been set elsewhere in the script or will have been arrived at earlier in the performance.

Section 3.3 Conditional Evaluations: if() and switchon

3.31 if ... then ... else

One of the main control flow mechanisms is given by the **if()** conditional branch, with or without the 'else' extension. It may be employed solely within a procedure, or in conjunction with the control-flow routine call it may be used to move from one procedure in the script to another.

The general form of **if()** is as follows:

```
if(some condition is TRUE ) {
     then execute the rules
     in this block
}
```

which may be extended by 'else' as follows:

```
if(some condition is TRUE ) {
     then execute the rules
     in this block
}
else {
     move to this block
     and execute these rules instead
}
```

The conditional evaluations use the comparative operators below

<	less than
>	greater than
==	is equal to
!=	is NOT equal to
<=	is less than or equal to
>=	is greater than or equal to
&&	logical AND
||	logical OR

The following are valid forms of conditional testing:

```
if(a < b)
if((x/3 >= z * 0.5) && (y != 0))
```

i.e., within the parentheses pairs of expressions have a relational operator between them.

Note that **while** also uses the same types of conditional evaluation as **if.**

3.32 switchon ... case statements

In order to avoid chains of **if..then..else..** statements, it is possible to use the **switchon ... case** construction.

The following script, 'swdemo.tv' should exemplify its use:

```
start()
{
        b = int(random(0, 3.999))
        while(1) {
                cnt += 1
                message "\n"
                print cnt, 4, 0
                switchon b {
                        case 0: {
                                message ": Case 0 chosen"
                                b = int(random(0, 3.999))
                        }
                        case 1: {
                                message ": Case 1 chosen"
                                b = int(random(0, 3.999))
                        }
                        case 2: {
                                message ": Case 2 chosen"
                                b = int(random(0, 3.999))
                        }
                        case 3: {
                                message ": Case 3 chosen"
                                b = int(random(0, 4.999))
                        }
                        default: {
                                message ": No action. There were "
                                print cnt, 4, 0
                                message " iterations.\n"
                                end
                        }
                }
        }
}
```

Section 3.4 Moving between procedures: call

The **call** routine may be used to move from one procedure to another. The keyword is used in the following manner:

```
call procedure_name()
```

Values may be passed to the procedure by including parameters in the parentheses of the procedure called, as in:

```
call refrain(5)
```

```
call update(x)
```

In the first of these examples, the value 5 is passed to the procedure 'refrain'; in the second example, the current value of cell 'x' is passed to the procedure 'update'.

Multiple values may be passed, as in the following script:

```
start()
{
     x = 9.3
     y = 77
     call proc1(x, 5, y)
}

proc1(f, g, h)
{
     probe f, g, h
}
```

For further examples of the use of **call** for moving between procedures, see the Manual entries for call, return, break and continue, and scripts in the appendices.

Section 4.0 Cells and Tables

Section 4.1 Global and Local cells

Cells are created by the *Tabula Vigilans* parser as it encounters them in the script. On creation, cells are automatically initialised to zero, until the script assigns a value to them. By default, all cells are global, i.e., they can be accessed from any procedure.

Local cells, private to a particular procedure, must however be declared. This means that the same cell name in another procedure will NOT be used, only the local one. Declarations of local cells at the head of the procedure are made as follows:

```
local cellname
```

Section 4.2 Indexing of Tables

Declaration of tables and their dimensions and sizes, in initialisation rules, has already been mentioned (see Section 1.3). Tables are collections of cells, and a cell can always be assigned a particular value in a table by indexing the table. It is important to note that table indices always run from zero to the table size - 1. So, if a table has been declared

```
table    PITCHES[18]
```

then the first value in the table PITCHES will be indexed as PITCHES[0] and the last – the eighteenth – as PITCHES[17]. In *Tabula Vigilans*, indices to tables are always made modulo the table size, so no harm will be done by indexing apparently outside the table. PITCHES[18], for example, will give the same result as PITCHES[0]. Even if the index goes negative, no harm will result: PITCHES[-1] is equivalent to PITCHES[17], PITCHES[-2] equivalent to PITCHES[16], etc. The following is enough to continually cycle round a table:

```
PITCHES[x++]
```

The index 'x' is used and then incremented each time it is accessed.

Section 4.21 Fractional Indexing

There is an alternative form of indexing in *Tabula Vigilans* which can prove useful: fractional indexing. Here the index is always taken as fractional, i.e., lying between 0 and 1. The notation used is the vertical line also used for OR: '|', which is placed around the index inside the square brackets:

```
PITCHES[|x|]
```

Here the index 'x' is taken to be a fractional index and will give the table value x of the distance along the table. If x is zero, then the first value in the table will result; if x is 1, then the last value in the table will result. If x is 0.25, then the value returned is that contained one quarter of the distance along the table. If x is greater than 1, then only the fractional part of x is used as the index. If x is negative, then the absolute value of x is used. The main advantage of fractional indexing is that it can be used to access arbitrary-sized tables.

Section 4.3 Table Pointers

Tables must be initialised at the head of the script (see Section 1.3). However, it is possible to use table pointers within a procedure simply by naming them, and assigning them to an already existing table. A table pointer might be used to point to different tables at different moments in a piece. The following shows the declaration of a table at the head of the file, and, later, within a procedure, a table pointer assigned to it:

```
table NOTES[7]
     [...etc...]
     [...now within a procedure...]

PITCHES = NOTES
```

It is worth noting that the fractional indexing mentioned in section 4.3 above is particularly useful when table pointers are used with the same index pointing to differently-sized tables. For example, suppose we have two tables A and B; A contains 13 values, B contains 29. In the following script, when the table pointer TP is changed from A to B, the fractional index 'ndx' will still point two-thirds along the larger table. This would be much trickier with integer indices:

```
Tabula Vigilans example to demonstrate
fractional indexing with table pointer

table A[7]
table B[13]

start()
{
    // Initialise the tables
    for(i=0; i<dimsize(A, 1); i+=1) {
    A[i] = i+1
    }
    for(i=0; i<dimsize(B, 1); i+=1) {
    B[i] = 11+i
    }
    // Set table pointer TP
    TP = A
    // Set fractional index
    ndx = 0.66
    // Show value of TP with fractional index

      message "Value of TP->A with fractional index 0.66\n"
      print TP[|ndx|], 5, 0
      // Change TP to point to table B
      TP = B
      message "\nValue of TP->B with fractional index 0.66\n"
      print TP[|ndx|], 5, 0
      message "\n"
}
```

Output:

```
Value of TP->A with fractional index 0.66
    5
Value of TP->B with fractional index 0.66
    19
```

The list of currently implemented keywords comprising primitive rules (opnames), control-flow mechanisms and mathematical functions is shown in the following Chart:

TV Keyword Buttons

abs	add_dec	alllocked	anylocked	arccosine	arcsine	arctangent
arg	argc	args	break	call	close_storefiles	cls
continue	compare	control-out	copy	copy_table	cosine	dimensions
dimsize	embed	end	exp	fail	fill_table	fold
for	gamma	gauss	generate	if	#include	int
int2string	interp_table	lim	lin	lintrans	local	lock
log	log10	loop	max	mean	message	messag1
midichord	midiecho	midiin	midiout	midiset	min	mouse
mult	mult_table	natlog	num2string	offset_table	perm	pitchbend
pop	power	print	probe	probi	rand	random
return	round	scale_table	schedule	seg	shift	showargs
sine	sort	sqrt	store_digits	store	storf	
stori	storefile	storstr	subst	sum	sum_table	swap
time	trigger	try	switchon	system	table	tangent
unlock	wait	while	xad	xar		

abs – Mathematical function

Converts the argument to its 'absolute' (i.e. positive) value. Examples:

```
x = abs(x) //Force x to become positive

y = abs(x) //Assign to y the positive value of x

z = abs(x-3.7) //Assign to z the positive value of (x-3.7)
```

add_dec – Rule

Conditionally add or decrement a value to a cell. There are four arguments: one output cell, and three input expressions, as follows:

```
out add_dec comp1, comp2, value
```

- If 'comp1' is less than 'comp2', then 'value' is added to 'out'
- If 'comp2' is less than 'comp1', then 'value' is subtracted from 'out'
- If 'comp1' is equal to 'comp2' then no action is taken.

Rule Return Status:

1 if action has taken place, **0** if no action has been taken.

alllocked – Rule

Test whether all of a list of cells are 'locked' (see 'lock' rule). Should be used in conjunction with **try** (see 'try' rule).

Examples:

```
probi try(alllocked a, b, c, d)

if(try(alllocked a, b, c, d) {
   ....
}
```

Rule Return Status:

1 if all cells are locked, **0** if one or more is unlocked.

anylocked – Rule

Test whether any of a list of cells is 'locked' (see 'lock' rule). Used in conjunction with **try** (see 'try' rule).

Examples:

```
probi try(anylocked a, b, c, d)

if(try(anylocked a, b, c, d) {
   ....
}
```

Rule Return Status:

1 if any cell is locked; **0** if none are locked.

arccosine – Mathematical Function

Also known as the 'inverse cosine'.

Example:

```
a = arccosine(b)
```

The input value to the function, b in the above example, which is NOT checked, should be a value between -1 and +1. The output will be a value between pi and zero.

arcsine – Mathematical Function

Also known as the 'inverse sine'.

Example:

```
a = arcsine(b)
```

The input value to the function, b in the above example, which is NOT checked, should be a value between -1 and +1. The output will be a value between -pi/2 and +pi/2.

arctangent – Mathematical Function

Also known as the 'inverse tangent'.

Example:

```
a = arctangent(b)
```

The input value to the function, b in the above example, may be from -largevalue to +largevalue. However, values below, say, -10 and above +10 will have decreasing effect. The output will be a value between -pi/2 and +pi/2.

arg – Mathematical function

Employ a value passed from the commandline. The number (n) of the argument will correspond to the nth value on the commandline, after the program name and the script name.

Example:

[On commandline]:

```
tv script 7.3 11 -3.05
```

[Within the script]:

```
a = arg(1)
b = arg(2)
c = arg(3)
```

Result:

'a' will take the value 7.3; 'b' the value 11, and 'c' the value -3.05

argc – Command line function

This function (previously undocumented) tests whether the correct number of arguments have been supplied on the command line. It is placed at the head of **initialise()**. It is customary to show the Usage and end running the script if the number of arguments is insufficient. The following example, which uses the ability (new in 1.5) of message to display a variable, illustrates this:

```
if(argc() < 5) {      //command line has less than 5 arguments
correctnumargs = 5
message "\nInsufficient arguments on command line\n"
message << "\nYou should have ", int2string(correctnumargs), " arguments on the command line, otherwise the script
will end or, better, show the Usage and end\n\n" >>
call usage()
end
}
```

The script **argctest.tv** illustrates the use of the function.

args – Command line string function

Pass a string (e.g. a filename) from the commandline. The number (n) of the argument will correspond to the nth string on the commandline, after the program name and the script name. It is possible to use this command line function with the script functions **storefile**, **storstr** and **fill_table**, and to display the string arguments with **showargs**.

Example:

[On commandline]:

```
tv script file.txt
```

[Within the script]:

```
storefile args(1)
```

Result:

The file named on the command line ('file.txt' in this case) will be used to store any data written from the script.

break – Control-flow

Break out of a 'for' or 'while' loop, with or without using a label. Without a label, control flow passes to the rule-lines immediately after the 'for' or 'while' loop in question. If the break is to occur across procedures (i.e., jumping out of one procedure and passing to another) the label mechanism should be used. A label is any convenient name which is placed immediately after the 'for' or 'while' keywords in order to identify them as the break location. Control then passes to the rule-lines immediately after the label, even if the label is in another procedure.

Example:

```
[In procedure a()...]

for wednesday (i = 0; i < 7; i+=1) {
        if(x > 32) {
        break
        // breaks out of this for()loop
        } //  no label reference required
        ....
}

[In procedure b()...]

if(t > 10) {
        break wednesday
        // break until after the 'wednesday'
        // label in procedure a()
}
```

call Control-flow

Call another procedure. When a performance first starts, control will be with the start() procedure. At some point it may be desirable to jump to another procedure. Suppose the user has named this second procedure watery(). Then the rule-line to do so will take the form:

```
call watery()
```

Moving back to the originating procedure can be effected by the **return** keyword (q.v.).

Values may be passed to procedures by placing cell names or values inside the parentheses of the procedure name. So

```
call watery(15.7)
```

will mean that the value 15.7 is passed to the procedure 'watery()'.

```
call watery(x)
```

will mean that the present value of 'x' in the originating procedure will be passed to the procedure 'watery()'. The receiving procedure 'watery()' may then assign a cell name, which in this instance will be local, to the value inherited from the originating procedure, by giving it a new name. E.g.:

```
watery(qq)
{
     if(qq < time_passed) {
          ..... [etc]
```

In this case the cell local to the procedure 'watery()' is called 'qq' and is assigned the value inherited from the call in the originating procedure (in the examples above, either 15.7 or 'x').

Note that multiple values may be passed, if required, as shown in the following short script:

```
start()
{
        x = 9.3
        y = 77
        call proc1(x, 5, y)
}

proc1(f, g, h)
{
        probe f, g, h
}
```

It is also possible to employ a label immediately after **call**, as in:

```
call saturn watery()
```

In this case the label ('saturn') may be used by a later procedure to return to this location. More than one call may use the same label – there is no conflict as the most recent label is used . (See Manual entry for **return**.)

close_storefiles – Control-flow

Close (all) open storefiles. This function may be used if the user wishes to re-use previously-written data later in a script. When a storefile is closed, data from the file may be used again by accessing it with 'fill_table'. The following script shows this process in embryo:

```
// 'numbers.tv' - show the use of 'close_storefiles'
table NUMBERS[4]

start()
{
        storefile "numbers.txt"
        for(i=0;i<dimsize(NUMBERS, 1); i+=1) {
                NUMBERS[i] = int(random(3, 17))
        }
        message "Table of 4 Numbers is "
        for(i=0; i<dimsize(NUMBERS, 1); i+=1) {
                print NUMBERS[i], 5, 0
                stori NUMBERS[i]
                storstr "\t"
        }
        message "\nNow closing storefile 'numbers.txt'.\n"
        close_storefiles
        NUMBERS fill_table "numbers.txt"
        perm NUMBERS
        message "Numbers have been read in again from the closed file,"
        message " and scrambled:\n\tthe order is now "
        for(i=0; i<dimsize(NUMBERS, 1); i+=1) {
                print NUMBERS[i], 5, 0
        }
}
```

cls – Control-flow

Clears the screen. Allows the user to remove previous message, print and probe data from the screen, either at the beginning of a performance, or even during performance.

compare – Rule

Prototype script line:

```
C compare A, B [,typeflag]
```

This rule compares two input tables, and writes into an output table either the values which are common to both, or values which appear in one input table only. If 'typeflag' is zero or is omitted, then the output table will put into the output table those values which are common to both tables. If 'typeflag' is non-zero, then the exclusive option will operate.

Suppose we have the following four tables:

A[8] contains the following MIDI notes:
62 65 70 71 72 63 66 67

while table
B[9] contains the following MIDI notes:
71 72 63 66 67 64 68 69 73

C[5] and **D[7]** have been declared but not yet initialised.

After the following script line:

```
C compare A, B, 0
  or
C compare A, B
```

then table **C** will contain
71 72 63 66 67

because 'typeflag' is **0**, meaning inclusive.

However, with 'typeflag' set to **1** – meaning exclusive – as in:

```
D compare A, B, 1
```

then table **D** will contain
62 65 70 64 68 69 73

which are the numbers appearing in either table but not in both.

Rule Return Status:

1 after the first and only action; thereafter **0**

continue – Control-flow

Continue with a **for()** or **while()** loop, with or without using a label. Without a label, control flow continues with the local **for** or **while**. A label is any convenient name which is placed immediately after the **for** or **while** keywords in order to identify them as the location for continue. Labels should be used if continue calls are made across procedure boundaries.

Example:

```
     [In procedure a()...]
for thursday (i = 0; i < 7; i+=1) {
     if(i < 4 || i > 4) {
          continue       // continues with this for() loop
     }                   // - no label reference required
     ....
}

     [In procedure b()...]

if(t > 10) {
     continue thursday   // continue with the 'thursday'
                         // label in procedure a()
}
```

control_out – Rule

Send a MIDI control message. There are three input cells: 'Channel' (0 - 15), 'Control-Number', and 'Value', respectively. The following example script uses **control_out** to continuously vary the volume of two MIDI channels:

```
start()
{
        if(first == 0) {
            midiset 0, 71
            midiset 1, 71
        volume = 7
            dur = 0.5
            pch = 60
        first = 1
        }

        if(try(midiout 0, pch, 100, dur, 1) > 0) {
            crdur1 = random(0.15, 0.45)
            pch = random(48, 72)
        }
        if(try(midiout 1, pch+1, 100, dur, 1) > 0) {
            crdur2 = random(0.15, 0.8)
        }
        dur = random(0.4, 2.0)
        a       lin     crdur1, 0, 110, 60
        b       lin     crdur2, 1, 110, 60
        control_out     0, volume, a
        control_out     1, volume, b
loop
}
```

Rule Return Status:

Always **1**

copy – Rule

Copies the value of an input cell into one or more output cells. Example:

```
a, b, c copy x
```

will copy the value of 'x' into a, b, and c.

Table pointers can also be assigned with copy, as in:

```
A, B copy C
```

NB Since **copy** is a rule, it should always be employed instead of assignments where cell-locking or unlocking is being employed. Mathematical assignments (of the type 'a = 5.6') will always override cell-locking.

Rule Return Status:

1 if copy has succeeded
0 if one or more copies has failed (e.g. if a cell has been locked).

copy_table – Rule

Copies the contents of one table to another. If the tables are of different sizes, then only values to the extent of the smaller-size table will be copied. For example,

```
BBB copy_table AAA
```

will copy the contents of table 'AAA' to table 'BBB'.

Rule Return Status:

1 if copy_table has succeeded in full
0 if the tables were of different sizes

cosine – Mathematical Function

Give the cosine of a value in radians; the output will thus always be a value between -1.0 and +1.0.

Examples:

```
a = cosine(2.17)
a = cosine(b)
```

Places the cosine of the input cell 'b' into cell 'a'.

dimensions – Mathematical Function

Return the number of dimensions of a table

```
// Script example:
table TAB[10][5][2]

start() {
        probi dimensions(TAB)
}
```

The script will print the number '3' to the console.

dimsize – Mathematical Function

Return the number of cells in a table dimension. There are two required arguments to **dimsize**: the table name (or table pointer-name) and the number of the dimension about which information is required. If, for example, a table has been declared

```
table AAA[8]
```

at the head of the script, then

```
x = dimsize(AAA, 1)
```

will assign the value 8 to 'x'.

Note that **dimsize** is an acceptable argument for the **int2string()** and **num2string()** functions (from Version 1.5), as in:

```
int2string(dimsize('SFILELIST, 1))
```

when used in a **message** or **messag1** statement.

dimsize can also be used with string tables in FOR loops to determine the end point of the loop:

```
for(i = 0; i <= dimsize('SFILELIST, 1); i += 1) {
```

embed – Rule

Embed two one-dimensional tables. The form of the rule is:

```
C embed A, B
```

The size of output table **C** should optimally be the sizes of input tables **A** and **B** multiplied together. (If smaller, embedding will be incomplete; if larger, the last table values will contain zeroes.) The data in the input tables should be equivalent, but the values in table **B** are made relative and then used to offset (by addition, **NB**) in turn each value in table **A** to create the output table **C.** While the rule runs in performance time, once the output table has been created it thereafter has no effect, even if values in the input tables change.

Rule Return Status:

1 after the first and only action; thereafter **0**

end – Control-flow

Ends performance. Under normal usage, the performance of *Tabula Vigilans* will continue until the user presses a key on the console. However, **end** is incorporated for use in those circumstances where it is desired that the performance end when a cell has reached a certain value. Example:

```
if(t >= 180.00) {
     end
}
```

exp – Rule

exp creates an exponential time-varying output moving between a minimum and a maximum value.

The prototype of the rule is:

```
out exp dur, [, direction [, scale_value1, scale_value2]]
```

'out' is the current value of the output
'dur' is the duration over which **exp** creates the exponential time-varying value
['direction'] – optional – is the starting direction:
0 (default) moves first from minimum to maximum
1 causes the first movement to be from maximum to minimum

NB Direction will be applied whichever order the values appear in 'scale_value1' and 'scale_value2'.

'scale_value1' and 'scale_value2' are the values to be supplied for the minimum and maximum. If these are omitted, minimum is **0**, maximum is **1**.

Rule Return Status:

1 at the end of each exponential segment reached
0 on every other occasion

fail – Rule

This rule is used for testing and debugging.

Rule Return Status:

Always **0**

fill_table – Rule

Fill a table from a textfile.
The table will previously have been declared at the head of the script and may be sized (e.g. TABLE[6]) or unsized (e.g. TABLE[]).
Fill_table may be used to fill a numerical or string table (the latter denoted by a prefixed ' character).

Example:

```
AAA fill_table "pitches" [, pos]
```

In this case table 'AAA' will be filled with values from a text file named "pitches" which is assumed to be in the current working directory. Unless filling a string table, the file should contain integer or floating-point values, separated by white space (i.e., spaces, tabs or carriage returns). Comments may be inserted after the double slash character //. If there are more values in the file than there are cells in the table, then only the first values to the size of the table will be read; if there are not enough values to fill the table, then the remainder of the table will contain zeros.

If the optional argument 'pos' is used, then this will be taken as the starting number. If 'pos' is **0** or **1**, then **fill_table** will read from the beginning of the file; this is the default condition. If 'pos' is any number greater than **1**, then the rule will begin the fill from that position.

```
AAA fill_table "intervals", 19
```

This example will fill table **AAA** starting with the 19th value in 'intervals'.

```
'SFILELIST fill_table "sndlist.txt"
```

In this case, a table of strings (text) is filled from a list of strings in a text file. Note that the text file should end on the final line and not put a carriage return after the final item. Otherwise, a blank item may appear when using the string table. Only single-dimension string tables are valid.

Currently there is no way to test the return status of **fill_table**.

fold – Rule

Time-based (multiplicative) embedding of one-dimensional tables. The form of the rule is:

```
C fold A, B
```

The output table 'C' will hold the values of table 'B' folded into table 'A' (compare rule **embed**). The data in the input tables are regarded as equivalent, but those in table 'A' are absolute (in seconds), those in table 'B' are relative, used to scale by multiplication the successive values of table 'A'. The size of the output table should be the sizes of the input tables multiplied together. While the rule runs in performance time, once the output table has been created it thereafter has no effect, even if values in the input tables change.

Rule Return Status:

1 after the first and only action; thereafter **0**.

for – Control-flow

Set up a loop counter, of the form:

```
for(i = 0; i < 12; i+=1) {
     A[i] = i+1
}
```

This simple example will set up a table 'A' with numbers 1 – 12 in it. However, the for loop, bounded by braces, may contain indefinitely many rule-lines.

gamma – Mathematical Function

Prototype:

```
g = gamma()
```

The result is a random number lying between **0** and **1**, exhibiting Gamma Probability Distribution characteristics. The statistical weighting is biased to the first half of the range, with a curve as shown below:

Gamma distribution curve

gauss – Mathematical Function

Prototype:

```
g = gauss()
```

The result is a random number lying between **0** and **1**, exhibiting Gaussian Probability Distribution characteristics. The statistical weighting is biased to the middle of the range, with a curve as shown below:

Gaussian distribution curve

generate – Rule

Prototype:

```
BBB generate AAA
```

This rule is designed to generate a first-order set of values in the output table by extracting adjacent intervals from an input table and in turn incrementing and decrementing them from each member of the input table. The output table is sorted in ascending order, and if the size of the output table differs from the optimum, the user is advised.

Rule Return Status:

1 after the first and only action; thereafter **0**

if – Control-flow

Conditional branch, with or without the **else** construct. The following syntax for the rule-file is required:

```
if(expression relation  expression) {
     ...other rule-lines...
     ...other rule-lines...
     ...other rule-lines...
}
```

The relation between the expressions should be one of the following:

Expression Relationships

<	less than
>	greater than
==	is equal to
!=	is NOT equal to
<=	is less than or equal to
>=	is greater than or equal to
&&	logical AND
||	logical OR

The following are examples of valid **if** clauses:

```
if(a == b)
```

if((a >= 3) && (b > a))

```
if(a*7 < b^0.3)
```

else may follow the 'if', in the following manner:

```
if(a >= 100) {
     b =  a % 7
} else {
     b = a % 3
}
```

#include – Initialisation

Incorporate another tv script within the present script.

Example:

[At the beginning of the current script:]

```
#include "otherfuncs.tv"
```

int – Mathematical Function

Return the integer value of a number (i.e the integer next lower than the number). Examples:

```
x = int(77.35)
```

will assign the value 77.00 to x, and

```
a = 55.32
b = int(a)
```

will assign the value 55.00 to b.

int2string – Type-conversion Function

Converts an integer to a string.

Examples:

```
int2string(12)
int2string(int_variable)
```

The function enables an integer value to be displayed by **message** / **messag1** or stored by **storstr**, using their second format in which one or more strings are enclosed within double-chevrons << >>. (See message or storstr for an example.)

The converted string cannot be assigned to a variable or an element of a string table. Thus the following will give syntax errors:

```
a = int2string(60)
'STRTABLE[0] = int2string(60)
```

interp_table – Rule

This rule interpolates a value between each corresponding value in two tables, and writes the result into a result table. An example rule-line is given below:

```
A interp_table B, C, 0.3
```

This means that each cell value in table **A** is calculated from the same index value of table **C** multiplied by 0.3, plus the same index value of table B multiplied by (1.0 - 0.3), or 0.7. If the rightmost input is itself a cell, when iterated while varying this value between **0** and **1** will result in a series of interpolated tables that move from table **B** to table **C**. Only the fractional part of the value of the right-most cell is taken as the rule argument. If tables of different sizes are given to the rule, then only the outputs corresponding to the smallest-size table are calculated.

Note – this rule does not apply to string tables.

Rule Return Status:

Always **1**

lim – Rule

Constrain a cell to lie within given limits.

lim has one output cell, and two inputs: 'limit-low' and 'limit-high'. The rule takes the form:

```
// limit cell 'a' to lie between limits
a    lim  -1.0, 1.0
a    lim  b, c
```

In the first of these, the meaning is that the value of cell 'a' is constrained to lie within the limits -1 to +1. In the second, 'a' is constrained to lie between the limits denoted by the values within cells 'b' and 'c'. The convention is that the values after the opname are 'low limit' and 'high limit' respectively, but in practice it does not matter - after all, the values within named input cells may change dynamically and 'cross over' by design. In either case, if the value in output cell 'a' lies outside the delimitation zone, it will be amended to the limiting value.

Rule Return Status:

1 if the limiting operation has succeeded
-1 if the operation has failed (because the output cell is locked)
0 if no action has been taken (because the output cell lies within limits).

lin – Rule

lin creates a linear time-varying output (i.e., moving between a minimum and a maximum value). The prototype of the rule is:

```
out lin dur, [, direction [, scale_value1, scale_value2]]
```

'out' is the current value of the output
'dur' is the duration over which lin creates a linear time-varying value

['direction'] - optional - is the starting direction:
0 [default] moves first from minimum to maximum
1 causes the first segment to move from maximum to minimum

[scale_value1 & scale-value2] are the values to be supplied as minimum and maximum respectively. If omitted, minimum is **0**, maximum is **1**.

Rule Return Status:

1 when lin is first called, and thereafter as the end of each linear segment has been reached.
0 at each other occasion.

lintrans – Rule

Implements a combined multiply and add in a single operation – a linear transformation of the type a = b * c + d. There is one output cell, and three input arguments. The prototype rule is:

```
result lintrans input, multiply_value, add_value
```

or, using the same identifiers as in the equation above:

```
a lintrans b, c, d
```

Example:

```
a lintrans b, 2.3, 1.7
```

If the output cell is identical to the input cell, the linear transformations will of course be incremental:

```
a lintrans a, 2.3, 1.7
```

Rule Return Status:

1 if the lintrans operation has succeeded
0 if the output cell is locked

local – Control-flow

Declare a cell to be local (private) to the current procedure. This has the effect that the same name can be used in other procedures with no conflict. Note that when a parameter is passed from one procedure to another, the cell receiving the parameter value is always local in scope.

Example:

```
solar()
{
    local sun  // the cell 'sun' is private to the procedure 'solar'
        ...
}
```

lock – Rule

Lock one or more cells.

All cells by definition contain values which may be treated as variables, i.e., they may be changed dynamically within the performance loop. However, there may arise occasions when the user does not wish the variable to be changed. In such cases the user may lock the variable against change, in effect making it a constant for the duration of the lock. (There is a complementary rule **unlock** which unlocks the cell again, and cells may be locked and unlocked at will during different segments of the performance loop.) A cell being locked will often change the meaning of a rule: examples of this are given in this Manual for **lim** and **sum**. Example rule-line:

```
lock a, b // lock cells 'a' and 'b'
```

It is important to understand that *rules will only obey rules*. Assignments will always over-rule rules. In the following block:

```
...
a = 136
lock a
a copy 245
...
```

since **copy** is a rule, it will obey the lock, and at the end of this block, 'a' will still equal 136. However, after the following:

```
...
a = 136
lock a
a = 245
...
```

'a' will now equal 245, because the assignment operator '=' has overruled the rule 'lock'.

Rule Return Status:

Always **1**

log – Rule

log creates a logarithmic time-varying output moving between a minimum and a maximum value.

The prototype of the rule is:

```
out log dur, [, direction [, scale_value1, scale_value2]]
```

'out' is the current value of the output
'dur' is the duration over which **log** creates the logarithmic time-varying value.

['direction'] - optional - is the starting direction:
0 (default) moves first from minimum to maximum
1 causes the first movement to be from maximum to minimum

NB 'Direction' will be applied whichever order the values appear in scale_value1 and scale_value2.

'scale_value1' and 'scale_value2' are the values to be supplied for the minimum and maximum. If these are omitted, minimum is 0, maximum is 1.

Rule Return Status:

1 at the end of each logarithmic segment reached
0 on every other occasion.

log10 – Mathematical function

Gives the logarithm, base 10, of any number.

Example:

```
a = log10(b)
```

loop – Control-flow

Return control-flow to the start of the current procedure.

max – Rule

Place in the output cell the maximum value of a number of input arguments.

Example:

```
a max b, c, d, e
```

If a cumulative result – over many iterations of a performance loop – is required, include the output cell in the input cell list, as in the following:

```
a max a, b, c, d, e
```

Rule Return Status:

1 under normal conditions
0 if the output cell is locked

mean – Rule

Place in the output cell the average, or mean, of a number of inputs arguments.

Example:

```
a mean b, c, d, e
```

The output cell 'a' stores the mean value of input cells b, c, d, e [....etc.].

Rule Return Status:

1 if the mean operation has been carried out successfully
0 if the output cell is locked

message – Rule
messag1 – Rule

Output to the console a single message string in one of two formats.

Example (Format 1):

```
message "This will be printed to the console"
messag1 "This will be printed only once"
```

These rules will output a message on the monitor during performance. For Format 1, the message to be displayed is placed within double quotation marks, but the latter will not be shown. The following special characters may be used within the string to produce the desired format:

```
\n (Newline)
\t (Tab)
\r (Return [to the beginning of the line, without Newline])
```

message will iterate the message as many times as it is called.
messag1 will print the message only the first time it is called.

From Version 1.5, it is now possible to contain multiple expressions, including variables, in the same message. The overall format is << ... , ... >>: string items separated with commas and placed within a container of double-chevrons.

Text including \n and \t is contained in double quotes in the usual way: "\n" and "\t", and "\nSome text" and "some text\n" continue to be valid (i.e., newlines included in the same quotes as the text string).

Example (Format 2):

```
intvar = 60
floatvar = 261.63
messag1 << "\nINTEGER VARIABLE: ", int2string(intvar), "\t", "FLOAT VARIABLE: ", num2string{floatvar), "\n\n" >>
```

which displays:

```
INTEGER VARIABLE: 60 FLOAT VARIABLE: 261.63
```
(along with the specified tabs and newlines).

Rule Return Status:

message Always **1**
messag1 **1** when the message has just been printed to the console, **0** thereafter.

midichord – Rule

This rule outputs a chord to the designated MIDI channel. Its prototype is:

```
midichord chan, CHORD, vel, dur [, num_notes [, arp]]
```

'chan' is the MIDI channel [0 - 15] on which the chord will be played
'CHORD' is the name of the TABLE which contains the notes of the chord
'vel' is the velocity of the notes, and
'dur' the duration of the chord

The number of notes can be altered by designating 'numnotes' – the default is 1.

'arp' is a flag to determine whether the chord should be arpeggiated:
1 = arpeggiate (over dur / num_notes)
0 = no arpeggiation

NB If the user needs to change the direction of arpeggiation, then a **sort** on the pitch table may be used prior to this rule.

Rule Return Status:

Positive number of notes immediately after launch
Negative number of notes thereafter, until dur has elapsed

midiecho – Rule

Output immediately (echo) a MIDI event. There are only three input parameters: Channel [0 - 15], MIDI note and Velocity.

Example:

```
midiecho 0, 60, 80
```

This will output middle 'C' on MIDI Channel 0 (i.e., synthesizer Channel 1!). Note that no duration value is given, so it is up to the user to turn off the note after the required duration by a matching command at some later time, e.g.

```
midiecho 0, 60, 0
```

The rule may be most useful when immediately responding to a **midiin** message, as in the following example:

```
start()
{
     c, n, v   midiin
               midiecho  c, n, v
loop
}
```

In this case the duration precisely matches that of the performed **midiin** rule. When you wish to output MIDI notes of a score-specified duration, use **midiout** (see below).

Rule Return Status:

Always **1**

midiin – Rule

If MIDI input data is available, **midiin** will collect it and store it in its output cells, of which there are three. The first is the MIDI Channel (0 to 15), the second and third differ according to the returned event type.

Example:

```
channel, data1, data2 midiin
```

Rule Return Status:

MIDIIN Data

0	No MIDI data available		
1	Note Event:	data1 = pitch	data2 = velocity
2	Pitchbend Event:	data1 = msb(low-res)	data2 = lsb(hi-res)
3	Aftertouch Event:	data1 = channel pressure	data2 = undefined – ignore
4	Controller Event:	data1 = controller no	data2 = value (msb)

Notes:

PITCHBEND:
the value can be 7-bit (lo-res) or 14-bit (hi-res)

- lo-res range: 0 127 – centre (no-bend) = 64
- hi-res range: 0 - 16383 – centre(no-bend) = 8192

CONTROLLER:
only the lo-res (8-bit) value is returned

- The second data byte for 14-bit hi-res controllers is dropped.
- For switches, ON = **127**, OFF = **0**
- Note that controller information may be instrument-specific – refer to the MIDI specification for information on controller numbers and messages.

midiout – Rule

Output a MIDI message.

Example:

```
midiout chan, note, vel, dur [, num_notes]
```

The input arguments are:

'chan' is the Channel number (0 15)
'note' is the MIDI note number (0 127)
'vel' is velocity (0 - 127) (i.e., loudness)
'dur' is the duration (in seconds)

[optional:]
'num_notes' is the number of notes to be output simultaneously – the default is 1.

If a rest is to be specified, then the MIDI note number (the value of the second input argument) should be made negative.

If fewer than 'num_notes' notes caused by this **midiout** rule are playing when the rule is executed, then a new note specified by 'chan', 'note', 'vel' and 'dur' is started, and the return status is the number of notes now playing. Otherwise there are at least 'num_notes' already playing and so the rule does not start a new note; the return status is minus the number of notes currently playing.

Thus

```
midiout 0, 65, 100, 1.5
```

will output one note event each time it is executed and the previous note has terminated.

```
midiout 0, note, 100, dur, 3
```

will keep three notes playing. Whenever a note finishes a new note will be added, using the current values of 'note' and 'dur'.

Any of the first four input parameters may be tables, in which case the first 'num_notes' table elements are used as the parameters to output 'num_notes' events.

Rule Return Status:

If the rule has succeeded in outputting any notes, the return status is the number of notes currently playing.

Otherwise the return status is minus the number of notes currently playing.

NB Executing the rule with 'num_notes' equal to zero returns minus the number of notes currently playing; a new note will never be launched in this event.

midiset – Rule

This rule may be used to set instruments to specific MIDI channels before performance. It may also be used to re-set instruments during performance if required. There are two input parameters: Channel number (0 15) and Instrument number (0 - 127).

Example:

```
midiset 0, 54
midiset 1, 63
```

Rule Return Status:

Always **1**

min – Rule

Output is the minimum value of up to nine input cells.

Example:

```
a min b, c, d, e
```

If a cumulative result – over many iterations of a performance loop – is required, include the output cell in the input cell list, as in the following:

```
a min a, b, c, d, e
```

Rule Return Status:

1 if the rule has been correctly applied
0 if the rule has failed (because the output cell is locked).

mouse – Rule

Prototype:

```
x, y mouse [scale1, scale2, scale3, scale4]
```

Output Cell 'x': Mouse 'x' coordinate
Output Cell 'y': Mouse 'y' coordinate

Gives two outputs, which are normalised screen coordinates of the mouse position (though on Atari the mouse cursor will be invisible!). If there are no input parameters, then the outputs are scaled between **0** and **1**. If there are input arguments, then these will scale the 'x' and 'y' values accordingly:

'scale1': Scale Value when mouse position is furthest left
'scale2': Scale Value when mouse position is furthest right
'scale3': Scale Value when mouse position is lowest
'scale4': Scale Value when mouse position is highest

Examples:

```
x, y      mouse
          probe     x, y

x, y      mouse     36, 96, -5.0, 5.0
          probe     x, y
```

Cell 'x' will represent the horizontal location of the mouse, and 'y' will represent its vertical location. In the first example, a value lying between 0 and 1 will result in 'x' and 'y'. In the second example 'x' will lie between 36.0 and 96.0, 'y' between -5.0 and +5.0.

Rule Return Status:

Always **1**

mult – Rule

Place in an output cell the result of multiplying the values of any number of input cells.

Example:

```
a mult b, c, d
```

The example given is equivalent to the equation

```
a = b * c * d
```

If output cell 'a' is unlocked, then its value will become the result of multiplying b, c and d. If it is locked, then each of the unlocked input cells is re-evaluated. If all cells are locked, then the rule return status records 1 or 0, depending on whether the two sides of the equation balance or not.

Rule Return Status:

1 if both sides of the opname are equal
0 if the **mult** operation has failed (because the output cell is locked)

mult_table – Rule

Multiply each value in a table by its corresponding index value in a second table, and places the result in the output table. Example:

```
A mult_table B, C
```

If the table sizes are unequal, then operations to the size of the smallest table only will occur.

Rule Return Status:

Always **1**

natlog – Mathematical function

Gives the logarithm, base e, of any number, where 'e' = 2.718281828....

Example:

```
a = natlog(b)
```

num2string – Type-conversion Function

Converts a floating-point number to a string.

Examples:

```
num2string(6.53)
int2string(float_variable)
```

The function enables an floating-point value to be displayed by **message** / **messag1** or stored by **storstr**, using their second format in which one or more strings are enclosed within double-chevrons << >>. (See message or storstr for an example.)

The converted string cannot be assigned to a variable or an element of a string table. Thus the following will give syntax errors:

```
a = num2string(6.53)
'STRTABLE[0] = num2string(6.53)
```

offset_table – Rule

Offset the contents of a table by a value, and write to an output table.

Examples:

```
A offset_table B, -1.0
A offset_table B, c
```

In these cases, each value in table B is offset, by the constant -1.0 and the current value of cell 'c' respectively, to produce the corresponding value in table 'A'.
If the table sizes are unequal, then operations to the size of the smallest table only will occur.

Rule Return Status:

Always **1**

perm – Rule

Randomly, or by step or skip, permute the contents of a one-dimensional input table.

Examples:

```
perm A
```

will permute the contents of table A.

```
perm A, 1
```

will permute table A by one step (ie two values in the table will be swapped). Successive calls with the same step value of 1 will cycle through all possible permutations of the table and a particular order will not repeat until all permutations have been iterated. The number of permutations is dependent upon the table size. The following gives the number of permutations for table sizes up to 8.

Permutations per Table Size

Table Size	Number of permutations
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320

As the number of permutations increases according to the factorial of the table size (**n!**), the use of step permutation with larger table sizes is not recommended. This stricture does not apply to random permutations.

```
perm A, skip
```

will permute table A by 'skip' steps. If skip is 2, every other step permutation will result, and the permutation cycle will be completed in half the number of steps. If 'skip' is otherwise a factor of the table size, the cycle will be correspondingly reduced. If 'skip' is a prime number not otherwise a factor of the table-size factorial, then an alternative path through the complete cycle of permutations will result. If 'skip' is negative, then the permutation cycle will be traversed 'backwards' - ie, with a table size of 4, **perm** A, -1 is equivalent to **perm** A, 23.

For reference, the following lists in four columns the size-4 table indices of a complete permutation cycle with a stepsize of 1:

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1
3 1 2 4
3 1 4 2
3 2 1 4
3 2 4 1
3 4 1 2
3 4 2 1
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
```

Rule Return Status:

Always **1**

pitchbend – Rule

Send a MIDI pitchbend message. There are two input parameters, containing the Channel (0-15) and pitchbend value respectively.

Bear in mind that the pitchbend range is normally set on the synthesizer itself. If this is set to one semitone, then the following will set notes on channel 4 to be a quarter-tone sharp:

```
pitchbend 3, 96
```

Rule Return Status

Always **1**

pop – Rule

Fractal algorithm which employs an output cell and an input value. Example rule-line:

```
a pop r
```

This algorithm is commonly referred to as the 'population' equation, from which the opname gets its name. It can be expressed:

$$a^{next} = r * a * (1.0 - a)$$

In this implementation of the algorithm, if the input cell 'a' happens to be larger than 1.0, then the reciprocal of this value is first substituted before the re-evaluation of the cell 'a'. As the value 'r' (normally a constant) is allowed to approach 4.0, the output results in more 'chaotic' behaviour. This value is not limited, so care should be taken by the user.

Rule Return Status:

1 if the algorithm has been successfully applied
0 if the output cell has been locked

power – Mathematical Function

Power function. Example:

```
a = power(b, c)
```

Output 'a' is 'b' to the power of 'c'. To square a value, use:

```
a = power(a, 2)
```

print – Rule

Print to the console the input argument. There are optional formatting fields.

print x [, format1, format2]

- 'x' is the cell whose contents are to be printed.
- 'format1' is the total number of screen characters to be taken in printing the value.
- format2' is the number of decimal places to be printed.
- The value is rounded to the last digit.
- If 'format1' and 'format2' are omitted, the rule defaults to 6 for format1 and 2 for format2.
- If 'format2' is given the value 0, then the number will be displayed as an integer.

In the following example, the cell 'x' has been initialised to 2.157042.

PRINT format options

Examples:	Result:
print x	2.16
print x, 4, 1	2.2
print x, 6, 2	2.16
print x, 8, 3	2.157
print x, 10,4	2.1570
print x, 10,5	2.15704
print x, 10,6	2.157042

Note that no carriage return or newline characters are output, so if the user wishes to incorporate line formatting, this must be done with message, e.g.

```
message "\n"
```

after print calls.

The **print** rule does not take a string parameter. Use **message / messag1** for this instead.

Rule Return Status:

Always **1**

probe – Rule
probi – Rule

Display the input arguments on the console. These rules are designed to show lists of cell values or arguments. Up to 10 can be displayed on an 80-character window, each pair separated by a tab. No newline is generated, so if the user wishes to see repeated calls to **probe** or **probi**, a message rule of the type

```
message "\n"
```

must be used after the probe line.

probe displays values to 2 decimal places.
probi displays values as integers.

Example:

```
probe a, b, c
```

Rule Return Status:

Always **1**

rand – Mathematical Function

Prototype:

```
a = rand()
```

The output is a random floating point number lying between 0 and 1

random – Mathematical Function

Prototype:

```
a = random(p, q)
```

The output is a random floating point number lying between the input arguments ('p' and 'q' in the prototype above).

return – Control-flow

Without a label, this means return to the last procedure. With a label, it means 'search for the most recent call with that label, and return to it. Example:

```
return saturn
```

means return to the latest call with the label 'saturn'. Note, therefore, that more than one call can refer to the same label the latest one is always used used, so there is no ambiguity.

round – Mathematical Function

Return the nearest rounded integer value of a number. Examples:

```
x = round(77.35)
```

will assign the value 77.00 to x, and

```
a = 55.2
b = round(a)
```

will assign the value 55.0 to b.

scale_table – Rule

Scale each cell in a table by a constant value to produce an output table. Examples:

```
A scale_table B, 3.3
A scale_table B, c
```

In these cases, each value in table 'B' is multiplied by 3.3 and 'c' respectively to produce the output values in table 'A'.

Note – this rule does not apply to string tables.

Rule Return Status:

Always **1**

schedule – Rule

Schedule a MIDI event for later performance. Example:

```
schedule channel, pitch, velocity, duration, delay
```

The scheduled event will occur 'delay' seconds later.

Rule Return Status:

Number of events scheduled for future performance.

seg – Rule

Create a linear time-varying output between values supplied by the user.

The prototype of the rule is:

```
out seg dur, value1, value2
```

'out' is the current value of the output

'dur' is the duration over which **seg** creates the linear time-varying value.

'value1' and 'value2' are the limits of the linear segment.

NB Contrast this rule with **lin**, which moves back and forth between minimum and maximum values.

Rule Return Status:

1 when **seg** is first called, and thereafter when the end of each linear segment has been reached.
0 on every other occasion.

shift – Rule

Shift contents of a one-dimensional table one place to the left [or right].

Prototype:

```
shift A [, direction]
```

If 'direction' is missing or is set to zero, then the contents of the input table are moved one place to the right, and the value of the last cell moves to the first. If 'direction' is non-zero, then the contents of the table are moved one place to the left, and the contents of the first cell gets moved to the last.

Rule Return Status:

Always **1**

showargs – Rule

Show (within a script) arguments invoked on the commandline.

Prototype:

```
showargs args(1)
```

The prototype will show the first argument string (after the script name) on the commandline.
If the script is named 'argstr.tv', and invoked with

```
tv argstr data.txt <RETURN>
```

then the prototype script line above will display

```
data.txt
```

Script example:

```
// 'strarg.tv' - Show commandline string arguments
start()
{
        message "Args(1) is "
        showargs args(1)
        message "\nArgs(2) is "
        showargs args(2)
        message "\nArgs(3) is "
        showargs args(3)
        message "\nArgs(4) is "
        showargs args(4)
}
```

If this is invoked on the commandline with

```
tv strarg data.txt Tuesday 6.37 Thursday <RETURN>
```

then the output will be:

```
Args(1) is data.txt
Args(2) is Tuesday
Args(3) is 6.37
Args(4) is Thursday
```

Note that the number in string argument(3) is shown exactly as it appeared, as a string.
Numbers can, however, also be displayed with 'print', as

```
print arg(3), 4, 2
```

in this case.

Rule Return Status:

None

sine – Mathematical Function

Give the sine of a value in radians; the output will thus always be a value between -1.0 and +1.0.

Examples:

```
a = sine(2.17)
a = sine(b)
```

Places the sine of the input cell in cell 'a'.

The following is a complete rule-file which will demonstrate **sine**.

```
// Demonstrate sine function.
start()
{
    pi2 = 6.28318530717958647692
    while(1) {
        a       sum     a, 0.001
        b = sine(a)
        probe   a, b
        a %= pi2
    }
}
```

sort – Rule

Sort a table into ascending or descending order.

Prototype:

```
sort AAA [, sort_flag]
```

If **sort_flag** is absent or is equal to zero, then the input table is sorted into ascending order. If **sort_flag** is non-zero, then the table is sorted into descending order.
NOTE: Currently only one-dimensional tables are supported with this rule.

Rule Return Status:

Always **1**

sqrt – Mathematical Function

Give the square root of a value. Example:

```
a = sqrt(b)
```

Places the square root of cell 'b' into cell 'a'.

store_digits – Rule
store – Rule
storf – Rule
stori – Rule
storefile – Rule
storstr – Rule

It is possible to store in a file values generated during the course of a performance. This enables, for example, re-runs of performances generated in real-time from statistical data.

The name of the **storefile** must be given in the form:

```
storefile "params"
```

If the file named already exists, then it will be overwritten.

The opname **store** has identical parameters to that of **probe**, and permits cell-values to be stored as floating point values to two decimal places in a single line of text in **storefile**. **stori** will store these values as integers, and **storstr** allows an ASCII string to be placed in the storefile.

The opnames **store**, **stori** and **storstr** are invoked thus:

```
store var1, var2, var3 .....[etc]
stori var1, var2, var3 .....[etc]
storstr "One string in quotation marks"
```

The following may be included within the quotation marks of **storstr**, for formatting purposes:

\n (Newline)
\t (Tab)

Multiple Output Files

From Version 1.33, it is possible to create more than one output file simultaneously. All files subsequent to the first are declared as follows:

```
storefile n filename
```
where *n* is an integer identifier (e.g. 1)

In the script, **store** and **stori** will **not** work with these integer-identified storefiles. Instead, the storage opname **storf** ('storefloat') should be used. With this opname it is possible to select in advance how many decimal places will be included in the storage, with the declaration

```
store_digits 4
```

Note that this is a global declaration - i.e., all variables stored with **storf** will have four decimal places.
If you wish to store integers with **storf**, simply declare '**store_digits 0**'.

Storstr does work with these files, however: simply add the integer identifier after the opcode '**storstr**':

```
storestr 1 "sometext"
```

The following is a complete script which demonstrates writing multiple output files:

```
// File 'stortest.tv'
start()
{
    storefile "pitchesA.txt"
    storefile 1 "pitchesB.txt"
    storefile 2 "pitchesC.txt"
        store_digits 0

    storstr "// PitchesA.txt\n"
    for(i=0; i<10; i+=1) {
        pitch = int(random(60, 73))
        stori pitch
        storstr "\t"
    }
    storstr "\n"

    storstr 1 "// PitchesB.txt\n"
    for(i=0; i<12; i+=1) {
        pitch = int(random(48, 59))
        storf 1, pitch
        storstr 1 "\t"
    }
    storstr 1 "\n"

    storstr 2 "// PitchesC.txt\n"
    for(i=0; i<24; i+=1) {
        pitch = int(random(72, 97))
        storf 2, pitch
        storstr 2 "\t"
    }
    storstr 2 "\n"
}
```

Multiple expressions in storstr

From Version 1.5, it is possible to contain multiple expressions, including variables, in the same store-string. For CDP users, this makes it possible to write command-lines and mixfiles, for instance.

The overall format is the same as Format 2 used in message / messag1, namely string items separated with commas and placed within a container of double-chevrons: << ... , ... >> .

Text including \n and \t is contained in double quotes in the usual way: "\n" and "\t", and "\nSome text" and "some text\n" continue to be valid (i.e., newlines included in the same quotes as the text string).

Storstr example (Format 2):

```
intvar = 1
storestr << "\nSoundfile ", int2string{intvar), "\t", 'SOUNDFILES[0], "\n\n" >>
```

For a single string-table cell, the format is not needed, as it is already a string: **storstr 'SOUNDFILES[0]** is acceptable.

Rule Return Status:

store, storf and **stori** – Always **1**
storefile, storstr and **store_digits** – None

subst – Rule

Place in an output table values drawn statistically from either of two input tables.

Prototype:

```
C subst A, B, fac
```

'fac' is a statistical substitution index factor, whose value lies between 0 and 1.

When 'fac' is 0, table 'C' copies table 'A'.
When 'fac' is 1, table 'C' copies table 'B'.
When 'fac' is some value between 0 and 1, each value of table 'C' is drawn from table 'A' or 'B' depending on the value of the factor. Since the choice is made statistically, repeated calls to subst even with the same substitution index is likely to produce a different output table.

Rule Return Status:

Always **1**

sum – Rule

Sum in the output cell the input arguments.

Example:

```
a sum b, c, d, e, f
```

If the output cell is unlocked, then cell 'a' will contain the sum of cells 'b' to 'f'. If 'a' is locked, then those cells 'b' to 'f' which are unlocked will be incremented (or decremented) equally to satisfy the equation.

```
a = b + c + d + e + f
```

If all cells are locked, then the rule return status will be either **1** or **0** depending whether the equation is satisfied.

sum may be used to increment or decrement a cell, by naming it as both input and output, as:

```
a sum a, -0.01
```

Rule Return Status:

1 if both sides of the opname sum are equal
0 if the output cell is locked and as a result the equation cannot be satisfied.

sum_table – Rule

This rule sums successive values in two tables to produce corresponding values in the output table. Example:

```
A sum_table B, C
```

In this case successive values in table 'B' are added to successive values in table 'C' to produce the successive values in table 'A '. If the tables are of unequal size, then only the minimum-size table values will result.

Note – this rule does not apply to string tables.

Rule Return Status:

Always **1**

swap – Rule

Swap the values of two cells. Cells within tables may be swapped. Examples:

```
swap a, b // Values in cells 'a' & 'b' are swapped
swap a, D[5] // Values in 'a' and D[5] are swapped
swap D[0], D[7] // Values in D[0] and D[7] are swapped
```

If a cell is locked, then no swap will occur.

Rule Return Status:

1 if the swap has occurred
0 if no swap has occurred

switchon/case – Control-flow

Conditionally branch to alternate code segments.

Example:

```
switchon a {
    case 0: {
        call rotate()
    }
    case 1: {
        call tumble()
    }
    case 2: {
        call undulate()
    }
    case 3: {
        call veer()
    }
    default: {
        dum = 0 // do nothing!
    }
}
```

system – System call

Creates a system call to an external program.
The program may be one normally accessible to the commandline interface (e.g. 'dir'), or it may be any program within the current path.
Redirection is supported.

From Version 1.5, there are two formats available for the command.
Format 1: the entire command is enclosed within inverted commas.
Examples:

```
system "dir >dirlist.txt"
      // Writes the current directory listing into a file 'dirlist.txt'.
system "dirsf C:\Sounds"
       // Lists the soundfiles in the named directory.
system "sndinfo len C:\Sounds\rock.wav"
       // Writes the length of the named soundfile to the console.
         // In the latter examples the path to the CDP executables
         // must be made known to the operating system.
```

Format 2 (from Version 1.5): multiple strings are placed within double-chevrons << >>, as also used in **message / message1** and **storstr**.
Example:

```
mode = 2
paramup = 12
system << "modify speed ", int2string(mode), " ", 'SFILES[0], " ", 'SFILES[1], " ", int2string(paramup) >>
```

This assembles and runs the CDP commandline: "modify speed 2 infile.wav outfile.wav 12" (where 'SFILES[0] and 'SFILES[1] were previously read from a textfile). The advantages of being able to run CDP commands from within *Tabula Vigilans* cannot be over-stated.

Note that when **system** is invoked, the entire control is passed to the called process. Any timing features of the script are interrupted until the called process returns control to TV.

table – Initialisation Rule

Create a (numerical) table of given dimensions and sizes. Examples:

```
table SOME[8] // 1 dimension, 8 cells
table MANY[20][2] // 2 dimensions, 20 x 2
table MORE[3][3][3] // 3 dimensions, 3 x 3 x 3
```

This initialisation rule occurs at the head of the script, outside any procedure. On initialisation, each table element or cell contains the value zero.

Typically, a loop might be used to fill the table. The [20] in the example above (table MANY) is the number of items in the table, accessed via the loop count. The [2] is the number of dimensions in the table, i.e., 2 columns. Thus we would index [k][0] for the 1st column and [k][1] for the 2nd column.

It is also possible to create a table without a fixed size. The table is declared with no number within the brackets and its size is determined when it is filled from a file, using **fill_table**:

```
table NEWTABLE[]
NEWTABLE fill_table "tabledata.txt"
```

However, multi-dimension tables must have their dimensions and sizes declared, as shown above.

String tables

String tables are identified by prefixing the character ' to the table name:

```
table 'SNDSTABLE[8]  or
table 'SNDSTABLE[]   [followed by a fill_table statement]
```

Multi-dimension string tables are not supported.

time – Rule

Set a real-time counter. Example:

```
t time
```

This places system time in cell 't', initialised to zero. On subsequent calls to **time** in the performance loop, the value of the cell is incremented in pace with real-time.

By processing this value, warped time may be created, e.g.:

```
start()
{
     a       sum     a, 0.0005
     t       time
     warp    mult    t, a
             probe   t, warp
     if(warp >= 100) {
          end
     }
loop
}
```

Rule Return Status:

Always **1**

tangent – Mathematical Function

The tangent function.

Example:

```
a = tangent(b)
```

Cell 'a' will store the tangent of the value in cell 'b'.

trigger – Rule

Set a trigger cell.

Prototype:

```
tr1 trigger timevalue
```

Result: 'tr1' is made positive each time 'timevalue' has elapsed

Example:

```
a trigger 0.5
if(a > 0) {
     pch sum pch, interval
}
```

Rule Return Status:

1 if trigger is operating normally
0 if the output cell is locked.

try – Mathematical Function

try is a special mathematical function which permits the user to ascertain the return status of a rule. It therefore takes the form

```
try(rule)
```

where **rule** is the complete rule-line used in a *Tabula Vigilans* script. The meaning of the return status is given at the end of each Manual entry for a Rule. Typically the return status means **1** for 'success' or 'normal operation' and **0** means 'failure' or 'abnormal operation' – but there are many exceptions and the user should check the Manual in cases of doubt. It is also important to realise that **try** applies only to rules – not to mathematical functions or to control-flow keywords.

It is possible to set a cell to store the result of **try**, as in:

```
x = try(midiout 0, note, 96, dur, 3)
```

and subsequently display it, as:

```
probi x
```

Or in an even more direct form:

```
probi try(midiout 0, note, 96, dur, 3)
```

unlock – Rule

Unlocks previously locked input cells. (See **lock**, above.) Example:

```
unlock a, b, c
```

Rule Return Status:

Always **0**

wait – Rule

Cause a complete stay of execution for the duration of the input cell, as in:

```
wait 2.3
```

Everything will be on hold for 2.3 seconds.

NB Since the normal *Tabula Vigilans* performance of a script is completely stopped by **wait**, this rule should be used with care. The only exception to the halt is that MIDI events already started and previously scheduled to terminate (as in **midiout** and **midichord**) will terminate correctly after their elapsed duration.

Rule Return Status:

Always **1**

while – Control-Flow

Set up a locally-executing loop dependent upon the evaluation of a test placed inside parentheses.

The form of the while clause is:

```
while(some condition is TRUE ) {
     execute this block of rules
     ....
     ....
}
[and if the condition is not true, jump over the block and come here ]
....
....[etc.]
```

The following

```
while(1) {
     ....
}
```

will always execute, because 1 is always true.

The test condition is evaluated using the usual relational operators – see Section 3.3. Examples:

```
while(x < 100) {
    ....
    ....
}

while(y > z && a == 0) {
    ....
    ....
}
```

xad – Rule
xar – Rule

Prototypes:

B xad A // eXtract Adjacent Differences
B xar A // eXtract Adjacent Ratios

These rules extract the differences (**xad**) or ratios (**xar**) between adjacent values in the input table, and write them to the output table, optimally wrapping beginning to end. If the tables are of different sizes, the output table will be either truncated or will contain zeros.

Examples:

Suppose we have a one dimensional input table A[4] as follows:

```
A: [3.31] [2.76] [7.13] [9.25]
```

then `B xad A` will place in output table B[4]:
B: [-0.55] [4.37] [2.12] [-5.94]

and `C xar A` will place in output table C[4]:
`C: [0.83] [2.58] [1.30] [0.36]`

Rule Return Status:

xad – Always **1**
xar – Always **1**

Demonstration Scripts

NB These demonstration scripts illustrate aspects of control-flow operations rather than musical uses of *Tabula Vigilans*. For the latter, see Richard Orton *Principles of Algorithmic Musical Composition* (forthcoming), or the disk of *Tabula Vigilans* script examples distributed with the CDP release.

```
////////////////////////////////////////////////////////////
// An example Tabula Vigilans script showing call() with
// parameters and break with and without labels (in this case
// 'fred' and 'jane')
////////////////////////////////////////////////////////////
start()
{
        message "Starting...\n"
        for fred (f = 0; f < 10; f+=1) {
                call sub1(f)
        }
        message "Fred is finished."
        while jane (j++ < 5) {
                call sub2(j)
        }
        message "\nJane is finished."
        for(k = 0; k < 10000; k+=1) {
                if(k == 273) {
                        message "\nKey number is"
                        probi k
                        message "\n\tIt's time to end....\n"
                        break
                }
}



sub1(x)
{
        if(x < 5) {
                message "This is Fred's time number"
                probi x
                message "\n"
        }
        else {
                break fred
        }
}

sub2(y)
{
        if(y >= 3) {
                break jane
        }
        else {
                message "\nJane is here!"
        }
}
```

```
////////////////////////////////////////////////////////////
//
// An example Tabula Vigilans script showing call() with
// parameters and continue with and without labels
// ('fred' and 'jane')
//
////////////////////////////////////////////////////////////
start()
{
        message "Starting...\n"
        for fred (i = 1; i < 10; i+=1) {
                if(i > 5) {
                        message "\nDone it"
                }
                else {
                        call sub1(i)
                }
        }
        message "\nFred is finished."
        while jane (j++ < 5) {
                call sub2(j)
        }
        message "\nJane is finished."
        for(k = 0; k < 10000; k+=1) {
                if(k == 273) {
                        message "\nKey number is"
                        probi k++
                        message "\n\tIt's time to end....\n"
                        continue
                }
                if(k == 317) {
                        message "Not quite: Key number now"
                        probi k
                        message "\nWe'll finish now!"
                        break
                }
        }
        end
}

sub1(x)
{
        if(x < 6) {
                message "This is Fred's time number"
                probi x
                message "\n"
        }
        else {
                continue fred
        }
}

sub2(y)
{
        if(y >= 3) {
                continue jane
        }
        else {
                message "\nJane is here!"
        }
}

// File "tab1", containing a set of 8 midinotes:
62 65 70 71 72 63 66 67

// File "tab2", containing a set of 9 midinotes:
71 72 63 66 67 64 68 69 73
```

```
////////////////////////////////////////////////////////////
// Tabula Vigilans script "compare.tv"
// demonstrating the use of 'compare'
////////////////////////////////////////////////////////////
table   AAA[8]
table   BBB[9]
table   CCC[5]

start()
{
    AAA fill_table "tab1"
    BBB fill_table "tab2"
    sect = 1
    CCC compare AAA, BBB, 0
    message "Section 1: Table A\n"
    while(1) {
        if(sect == 1) {
            if(xx > 0) {
                a = AAA[andx++]
            }
            if(count > 24) {
                count = 0
                sect = 2
                message "Section 2: Table B\n"
                wait 2.0
            }
        }
        if(sect == 2) {
            if(xx > 0) {
                a = BBB[bndx++]
            }
            if(count >= 27) {
                count = 0
                sect = 3
                message "Section 3: Table C: Intersection of Table A & B\n"
                wait 2.0
            }
        }
        if(sect == 3) {
            if(xx > 0) {
                a = CCC[cndx++]
            }
            if(count >= 20) {
                end
            }
        }
        xx = try(midiout 0, a, 64, 0.2, 1)
        if(xx > 0) {
            count += 1
        }
    }
}


////////////////////////////////////////////////
//
// Demonstration Tabula Vigilans script for 'copy'
//
////////////////////////////////////////////////
start()
{
        d = 7.73
        message "At start: a, b, c, d are \t"
        probe a, b, c, d
        message "\nNow copying value of d into a, b, c:"
        a, b, c  copy d
        message "\nNow, cells a, b, c, d are \t"
        probe a, b, c, d
        message "\n"
}




////////////////////////////////////////////////
//
//  Demonstration Tabula Vigilans
//   script for table copying
//
////////////////////////////////////////////////
table TABLE1[4]
table TABLE2[4]
!--

 -->
start()
{
   TABLE1 fill_table "input1"


   message "Input Table is:\n"
   probe TABLE1[0], TABLE1[1], TABLE1[2], TABLE1[3]
   message "\n"
   TABLE2 copy_table TABLE1
   TABLE1[0] = 77
   TABLE1[1] = 78
   TABLE1[2] = 79
   TABLE1[3] = 80
   message "Output (copied) Table is:\n"
   probe TABLE2[0], TABLE2[1], TABLE2[2], TABLE2[3]
}
```

```
//////////////////////////////////////////
//
// Demonstration Tabula Vigilans script:
//   use of dimsize
//
//////////////////////////////////////////
table TAB[10][5]


start()
{
	message "dimensions(TAB)  = \t"
	probi dimensions(TAB)
	message "\n"


	TAB1 copy TAB


	message "dimensions(TAB1) = \t"
	probi dimensions(TAB1)
	message "\n"


	call x(TAB)
}


x(T)
{
	message "dimensions(T) = \t"
	probi dimensions(T)
	message "\n"
	message "Size of dimension 1 is\t"
	probi dimsize(T, 1)
	message "\n"
	message "Size of dimension 2 is\t"
	probi dimsize(T, 2)
	message "\n"

	T[0][0] copy 11.23
	T[2][1] copy 57.82
	T[8][2] copy 93.77

	for(a = 0; a < 10; a += 1) {
		T[a][a+1] copy a*a
	}
	message "Table initialized\n"
	for(a = 0; a < 10; a += 1) {
		call prrow(T, a)
	}
}


prrow(T, row)
{
	for(cnt = 0; cnt < dimsize(T, 2); cnt += 1) {
		print T[row][cnt]
		message "\t"
	}
	message "\n"
}
```

```
////////////////////////////////////////////////////////////
// argctest.tv (AE - 01 May 2023)
// Command line: tv argctest.tv 60 67 60 90 "midipchlist.txt"
// Leave out one of the numbers to activate the argc message
// The number of commandline arguments found can be saved to a
// variable.
// The argument count begins after the call to tv, which is
// actually arg(0)
////////////////////////////////////////////////////////////

table PCH[]

start()
{
  if(init == 0) {
    call initialise()
  }

  messag1 "\nThe contents of the first four arguments are:\n"
    probi pchlo, pchhi, vello, velhi

  messag1 "\nThe contents of the MIDI pitches file are:\n"
    for(i = 0; i < dimsize(PCH, 1); i += 1) {
      probi PCH[i]
      message "\n"
    }
}

usage()
{
message <<"USAGE: tv15 60 67 60 90 midipchlist.txt", "\t", "(5 arguments after 'tv' )\n" >>
}

initialise()
{
  message "Command line with 5 arguments\n"
  if(argc() < 5) {      //test by leaving out an argument
    correctnumargs = 5
    message "\nInsufficient arguments on command line\n"
    message << "\nYou should have ", int2string(correctnumargs), " arguments on the command line, otherwise the script will end or, better, show the U

    call usage()

    end
  }

  pchlo = arg(1)
  pchhi = arg(2)
  vello = arg(3)
  velhi = arg(4)


  PCH fill_table args(5) //note 'args' for a textfile containing
                         //  a list of pitches (numbers)
                         //  -- or other data
                         //The '5' indicates it is the 5th argument.

  init = 1
}
```