T N T S C R I P T S
General Documentation
(as of IX.2009)

CONVENTIONS

Throughout this document, regular TNT commands are shown in CAPITALS, macro commands in **bold**, internal variables with *italics*, and user variables as underlined.

ACTIONS

In the vast majority of cases, actual actions are done with regular TNT commands (those listed under "help;"). The idea in scripts is using specific macro commands, which allow to automatically control the actions that are executed (or making specific calculations with values obtained from actions). This handout concentrates on the use and syntax of scripting commands only. For the use of general (regular) commands of TNT, the user should refer to the general documentation.

COMMANDS FOR SCRIPTS/MACROS

Commands for scripts are those listed under "help+;". They allow making decisions, accessing internal variables of the program (e.g. how many trees are held in memory, what's their length, etc.), and assigning values to variables defined by the user.

The internal variables of the program can be accessed only within the context of the commands for macros; they are read-only values.

The variables defined by the user can be named; writing the name (or number) of the variable within quotes is equivalent, *within any context*, to writing the value that has been assigned to the variable. User variables can be assigned values (or, rarely, strings), with the (macro) command **set**.

The user variables provide the bridge between macro commands or expressions (decisions, internal variables) and the regular TNT commands (actions). Keep in mind that internal variables can only be recognized from within macro commands, not from regular TNT commands. An example: the expression *ntrees* corresponds to the internal variable "number of trees minus one". The (regular) KEEP command retains as many trees as specified, so that "KEEP *ntrees*" might be used to discard the last tree in memory –except that KEEP isn't supposed to recognize expressions for internal variables. The command **set** (a macro command) does recognize such expressions, and then it can be used to assign the number *ntrees* to an internal variable (say, number 0), which (if enclosed within quotes) can subsequently be recognized within any context as representing the

number of trees minus one:

```
set 0 ntrees ;
KEEP '0' ;
```

The following example also illustrates a very simple decision-making process (showing in **bold** the macro commands, *italics* the internal variables, underlined the user variables, and CAPITALS the regular TNT commands):

```
PROC %1 ;                /* read data set, from first arg ... */
var: success ;           /* declare user variable  */
MULT ;                   /* make a search... */
if ( length[ 0 ] < 1000 )  /*  check the result  */
     set success 1 ;
else  set success 0 ; end
if ( 'success' )           /*  report result...  */
     QUOTE The search found less than 1000 steps ;
else  QUOTE The search failed to find less than 1000 steps; end
PROC/;            /*  close file with instructions */
```

(the intermediate step of placing the value of the length comparison in a user variable has been added only to make the example clearer, as it is obviously unnecessary).

EXPRESSIONS

General syntax is similar to that in C. Precedence is checked from left to right (the parser works by calculating the values as it reads the expressions). Operators: arithmetic (+ - * /), comparisons (== > < >= <= !=), logical (&& || !), and bitwise operations (& | ^). When a number is expected, within the context of a macro command, enclosing a series of numbers (0-29) in curly braces is equivalent to typing the integral number that corresponds to that bit set (that is, { 0 3 4 } = $2^0+2^3+2^4$ = 1 + 8 + 16 = 25). Bitwise representation is used (as in other parsimony programs) to store sets of states at internal nodes or polymorphic terminal taxa; the expressions that access state-sets (such as *states*, see under "help+states;") return sets of states as sets of bits.

The legal operations and comparisons are:

```
a == b     if a equals b, value is 1 (0 otherwise)
a > b      if a is greater than b, value is 1
           (same for <)
a >= b     if a is greater than or equal to b, value is 1
           (same for <=)
a != b     if a is different from b, value is 1
a && b     if conditions a and b fullfilled, value is 1
a || b     if conditions a or b fullfilled, value is 1
!a         if condition a is not fullfilled, value is 1
```

a & b	if a bit is 1 in ***a*** and in ***b***, it is 1 in the result
a | b	if a bit is 1 in ***a*** or ***b***, it is 1 in the result
a ^ b	if a bit is 1 in a or b (but not both), then it is 1 in the result

A "condition" may consist of just the numerical value of a number. For example, the variable success in the example above, might have been checked as

```
if ( 'success' == 1 ) ...
```

but, given that success itself can take values 0 (zero, "false"), or different from 0 ("true"), this is unnecessary, and identical to the more economical expression used in the example:

```
if ( 'success' ) ...
```

Note that a negative number is different from 0, and therefore is also evaluated as "true".

Also, the value returned by a logical comparison is either 0 or 1, which sometimes facilitates writing more compact instructions. For example, placing in a variable numlongtrees the number of trees that are longer than (say) 1000 steps might be done as:

```
loop 0 ntrees
     if ( length[0] > 1000 )
          set numlongtrees 'numlongtrees'+1 ;
          end
     stop
QUOTE There are 'numlongtrees' above 1000 steps ;
```

But the same result would be achieved by the more compact:

```
loop 0 ntrees
     set numlongtrees += ( length[0] > 1000 ) ;
     stop
QUOTE There are 'numlongtrees' above 1000 steps ;
```

INTERNAL VARIABLES

See list under "help+;". They allow to recognize dimensions of the data (taxa, characters, number of trees in memory, etc.), tree statistics (length, number of branches, number of taxa, etc.), making lists of nodes for given trees (up and down), identifying a common ancestor of some nodes in a tree, character/taxon activities, etc. etc. etc.

DECISIONS

Made with commands **if**, **else** and **end**. Every **if** has to be matched by an **end** (or, optionally, an **else**), and every **else** has to be matched by an **end**. General form:

```
if ( condition )
     actions A, decisions A ...
else
     actions B, decisions B ...
     end
```

If the condition that follows the **if** is different from 0, then actions/decisions A are executed, otherwise actions/decisions B are executed. There can be any number of nested **if**/**else**'s.

LOOPING

Loops are used to execute repetitive actions. The general form is:

```
loop =name X+Y Z
     actions, decisions ...
     stop
```

where X, Y and Z are integral numbers, and "name" is the name (optional) which the loop receives. The loop is executed from X (inclusive) to Z (inclusive), every time increasing value in Y (this is optional; if the "+Y" is ommitted, then loop value is increased in 1 every time). Every time a loop of level N is executed, if the expression #N is found within the instructions for the loop, then this is replaced by the value corresponding to that loop iteration. If the loop has been named, then "#name" produces the same effect. If there are nested loops, the first (outermost) is #1, second is #2, etc. Within the loop, the following commands are recognized: **endloop** (terminates), **continue** (moves onto the next cycle), and **setloop** N (re-sets loop to value N and continues running). Loop numbering is specific for every input file (i.e. outermost loop in a file is always #1, regardless of loops in previous input files); thus, the only way to access the value of a loop from a previous input file is through the use of a loop name.

Other commands that execute repetitive actions, for specific purposes, are:

sprit N.- executes all the actions between **sprit** and **stop**, for every SPR rearrangement of tree N. Commands recognized from within **sprit** are: **resetswap** (replaces tree N by the current rearrangement, and begins to rearrange the new tree), **continue**, and **endswap**.

tbrit N.- as **sprit**, but doing TBR rearrangements.

RESAMPLE, QNELSEN, SECTSCH.- These commands also execute instructions within a loop. In the case of **resample** and **qnelsen**, a single level is permitted (i.e. a **resample** cannot be nested within another **resample**); in the case of **sectsch**, there can be multiple levels. The instructions to be executed within the loop are specified within square brackets; the instructions are treated as if they were a new level of input file, which means that loop numbering starts again from 1, and the execution of a given cycle can be interruted with "PROC/;" (but note that a PROC/ is not required at the end). In the case of **resample**, the data in every cycle are automatically resampled (according to the type of resampling required by the user, see "help resample;"). After executing all the instructions for the cycle, the trees that are left in memory (presumably, resulting rom analyzing the resampled data with search commands or routines to the choice of the user) rae used to calculate a strict consensus, and at the end the majority rule tree, or the frequency difference consensus, are calculated. ***Qnelsen*** works in a similar way, but without resampling (so that it allows quick consensus estimtations). ***Sectsch*** creates reduced data sets, for running sectorial searches; in every cycle, the data received will correspond to a reduction of the tree, to effect a sectorial search with routines that can be defined by the user. Every instance of ***sectsch*** inherits a (single) tree in memory, corresponding to the current resolution of the tree used to partition the data. Every time a cycle is completed, **sectsch** automatically selects one of the best trees that have been left in memory, and reinserts it into the main tree. In the case of ***sectsch***, if the routine to analyze each sector is not specified, the default algorithms for sector analysis are used. In the case of **qnelsen** or **resample**, if the routine to analyze every cycle is not specified, the last routine used (or the default one) is used.

Iterrecs.- **Iterrecs** allows generating (and accessing) every one of the most parsimonious reconstructions, for a specified tree and character. The syntax is:

```
iterrecs tree character variable
     ... instructions...
     ...
     ...
      endrecs ;
```

the variable must be an array, with enough cells to contain at least as many values as nodes there are in the tre. In every cycle, the variable will contain values corresponding to the node states for that particular reconstruction. Within **iterrecs**, the command **killrecs** (which interrupts the whole process) is

recognized. It is possible to have multiple levels of **iterrecs**. Additional details (like, how to use ancestor-descendant differences instead of the states themselves, and how to force presence or absence of a given state to a node, so as to find the best reconstruction which either has, or doesn't have, that state) can be seen under "HELP **iterrecs;**". The instructions for **iterrecs** are treated as a new level of input file (so that they can be interrupted with "PROC/;").

Travtree.- This command allows travelling through a tree, visiting the nodes in a specified sequence. Syntax is:

```
travtree type tree node variable
     ... instructions...
     ...
     ...
     endtrav ;
```

the variable is a simple variable (not an array), which, in every cycle, will contain the number of node being visited. "Type" can be one of the following: 1) ***down***, visits the nodes of the subtree corresponding to the specified node (root = entire tree), in a down-pass; 2) ***up***, like the previous one, but in an up-pass (up can be followed by the string ***terms***, if you want the terminals corresponding to the subtree to also be visited; otherwise only internal nodes are visited); 3) ***below***, which travels from the specified node towards the root, and 4) ***path***, which travels between two nodes (it requires specification of two nodes instead of one). If the variable is preceded by a minus sign (-), then the first element in the list is skipped. Within ***travtree***, the commands ***skipdes*** (it has an effect only in the type ***up***, and makes subsequent cycles to skip the descendants of current node) and ***killtrav*** (immediately finishes the trip) are recognized.

Combine.- This command allows enumerating combinations, in a loop. Syntax is:

```
combine X min/max varname
     ... instructions...
     ...
     ...
     endcomb
```

which enumerates the combination of min out of X elements, then min+1 out of X, min+2 out of X, ... max out of X. If "/max" is ommited, then max = min. The elements are written to variable "varname" (must be an array), and *listsize* equals number of elements minus 1 (keep in mind that *listsize* can be subsequently modified within the loop, make sure to store value at the beggining of loop if this is so).

For example, assume "array" is a proper array; then

combine 4 2 array QUOTE 'array[0-*listsize*]' ; **endcomb** ;

will produce:

```
0 1
0 2
0 3
1 2
1 3
2 3
```

as output.

USER VARIABLES

DECLARATION
User variables are declared as:

var: *name_a name_b name_c*[*dims_for_c*] ;

this declares *name_a* and *name_b* as simple variables, and an array *name_c* (of a number of values, or cells, equal to *dims_for_c*). If several declarations are repeated (i.e., new instances of **var:**), the new declaration will place the first declared variable in the memory area that is contiguous to the last variable declared in the previous instance of **var:**. It is possible to un-declare variables (for example, to alleviate the use of memory by the macro system), with **var – name;** or **var – N** (this eliminates from the list all the variables that had been declared after variable called ***name***, or numbered as N, inclusive). If no name or number is specified, then all the variables that had been declared in the current input file are un-declared. For alternative ways to declare variables, see documentation of TNT. Variable declaration is internal of every input file; every time an input file, al the variables that had been declared within that input file are automatically eliminated. If a variable is declared with the same name that had been used in a previous input file (not yet closed, from which the current one was called), then the name refers to the variable in the current file (variables in other input files can be accessed only if they have a different name).

SETTING VALUES

To asign a value to a variable, the **set** command is used, followed by the name (and cell, if an array) of the variable, and an expression (to indicate the value that the variable will take). A few expressions (=internal variables) automatically transfer values to array (such as **freqlist**, **freqdlist**, **bremlist**, **uplist**, **downlist**, **randomlist**; see help+ for details), case in which the

variable given to **set** has to have enough cells available. The expression **states** also can, optionally, return values as uni- or bi-dimensional arrays (replacing the number of characters, or taxa, or both, by a period). As for the expression itself, if it is replaced by ++ the value of the variable is increased in one, with -- it is decreased, with +=X it is increased in X, and with -=Y it is decreased in Y.

It is possible to store a string in a variable (case in which, you have to be careful not to overwrite variables adjacent in memory; the program does NOT check for this error), giving after the name of the variable the symbol "$" followed by the string (strings are accessed in the same way as they are set, see below).

Last, it is also possible to assign values to a complete array, with the **setarray** command.

ACCESS

Once user variables have been assigned a value, it is possible to use that value in comparisons, calculations, etc. Enclosing the name (or number) of the variable within single quotes is equivalent to writing the value that has been assigned to the variable, *in any context*. It is this feature which allows using these variables in regular TNT commands.

As example, consider a case where we want to select a taxon at random to be deactivated. The expression *getrandom* is recognized exclusively within the context of the macro commands (i.e. those listed under help+), and therefore we could NOT do something like:

```
TAXCODE - getrandom[0 ntax] ;
```

since the TAXCODE command is not a macro command and it expects as arguments(s) nothing more than numbers, taxon names, or taxon groups. This would have to be done as follows:

```
var: thetax ;                      /* declare variable... */
set thetax getrandom[0 ntax] ;     /* ... and assign to it a
                                      random number, between 0
                                      and number of taxa - 1 */
TAXCODE - 'thetax' ;               /* deactivate taxon  */
```

The number of decimals with which the value of the variable is read is (by default) the one that has been set with **macfloat** N (default = 5). It is possible to give a specific format to the number (useful for creating formatted output, see below, under FORMATTED OUTPUT).

In the case of arrays, it is possible to convert the variable into a series of values automatically. Thus, if the variable called (for example) listaxa contains numbers 4, 8, 3 , 12, and 15 as its first 5 values (cells 0-4), the expression 'listaxa[0-4]' is equivalent to writing "4 8 3 12 15" (for example, TAXCODE - *'listaxa[0-4]'* will deactivate those five taxa). If the second

number is followed by the symbol & and a number N, then character ASCII N is used as separator of the numbers: 'listaxa[0-4 &43]' is equivalent to writing "4+8+3+12+15". This is useful to create lists containing a number of elements that is unknown before running the script. The following script lists the taxa which have been deactivated:

```
var:
   numelements
   dalist[ (ntax + 1) ] ;
set numelements 0 ;
loop 0 ntax
      if ( isactax [ #1 ] ) continue ; end
      set dalist [ 'numelements' ] #1 ;
      set numelements ++ ;
      stop
macfloat 0 ;
QUOTE A total of 'numelements' taxa are currently inactive: ;
QUOTE 'dalist [ 0 - ('numelements'-1) ]' ;   /* we dont know
                                              ahead of time how many
                                              numbers we print! */
PROC/;
```

If instead of enclosing the number/name of the variable within quotes, it is preceded by the symbol $, then the variable is interpreted as the string the begins at that position. This can also be applied to some special cases (the meaning of which, I hope, will be obvious to the reader): $dataset, $taxon N, $character N, $state C S, $ttag N, $host N.

PLOTS/CORRELATION

It is also possible to produce (veeery simple) plottings of the values in one (or two) array(s), with the **"var+"** option. The option **"var&"** calculates a lineal regression between two variables (the values are stored in the internal variables ***regr***, ***regalfa***, y ***regbeta***, of obvious meaning). See "help+var" for details.

The ***maketable*** command, through the specification of an array (or two), allows displaying the values of the array in table format (a double table, in case two arrays are specified). See "help maketable;" for details on how to use this command.

PROGRESS INDICATOR

The command **"progress** *done todo text;*" (where *done* and *todo* are numbers) shows what proportion of *todo done* represents, in the form of a progress bar. The progress bar also automatically

checks for user-interruptions, so that the macro routine can be interrupted. To use the **progress** command, it is necessary that the "report" option (REPORT command) be off. Once the task is concluded, the progress bar must be closed with **progress/.**

MEMORY FOR MACROS

The memory used for macros is separate from the memory used for data, trees, etc. The user variables occupy memory for the macros, and all the copying of the instructions of every level of looping (and subsequent expansion of the corresponding expressions in every cycle) is also done in a region of memory accessible to the macros. If you need to increase the number of available variables (default = 1000), or maximum loop nesting (default = 10), the command **macro * L V** can be used (where L = max loops, and V = max. variables). The total amount of RAM to be used by the macros can be changed with **macro [K** (where K = number of kylobytes to assign to the macros, default = 100). The maximum nesting for input files can be changed with the MXPROC command. It is possible to store user variables as int's, instead of double's, case in which every cell of a user variable will occupy 32 bits (with **macfloat-**), instead of 64 (with **macfloat=**, the default); this can be used to diminish the amount of RAM that a given routine will need.

INPUT REDIRECTION

NORMAL

To begin reading instructions from a file, the command PROCEDURE can be used, followed by the name of the file to parse. The command RUN does the same, but it reads all the arguments following the name of the file (max. 32), until it finds a semicolon:

run myfile arg1 arg2 arg3;

then, within file ***myfile***, the expression %1 is equivalent to ***arg1***, etc. (%0 is equivalent to the name of the file itself, that is, ***myfile***). When TNT finds a command it doesn't recognize, it checks to see whether a file with that name, and extension *.run, exists in the current directory, and if so, it reads and executes the file (taking arguments).

The file with instructions is executed until the command PROCEDURE/ (i.e. PROCEDURE with a slash) is found. It is possible to have nested input files (the default maximum is 15, but this can be changed with MXPROC). Alternativelly, the file can be closed with the **return** command, followed by a number. This closes the file and writes in the internal variable *exstatus* ("exit-status") the exit value.

"FUNCTIONS" (GOTO)
With the command:

goto myfile tagname arg1 arg2 arg2 ;

instructions are taken from file ***myfile***, beggining at the point marked with ***label tagname***. The arguments are as in the preceding cases. A default "target" file can be defined for goto, with **goto = *defaultfile*** (and note: **goto = %0** defines as default the file itself). After having defined a default target, the name of the file after the **goto** command is skipped (specifying just the ***tagname***). After concluding execution of the instructions in ***myfile***, the instructions that had not been read in the file from where ***myfile*** had been called continue being read (that is: this is different from the goto in most programming languages, because it executes and returns to the calling point, acting more like a function). This can be used to make instructions files simpler and cleaner, placing the necessary routines under different ***labels***.

AUTOMATIC INPUT REDIRECTION (@@)
At (almost) any point during execution, if a double "at" (@@), is found in the input, it is interpreted as a **goto** (with the end of the file indicated also as double at, instead of PROC/). The difference with goto is that automatic redirection allows replacing expressions within executions of commands, by substituting the contents of the file ***textfile*** instead of the @@textfile. Thus, the following code:

quote Sorry. An error occurred. @@textfile message1 $dataset ; ;

(note that *two* semicolons are needed here, since the first one indicates termination of the chain of arguments for input redirection, and it is ignored by the QUOTE command; the second semicolon terminates the QUOTE). If the file ***textfile*** contains:

label message1
Cannot analyze file %1.
@@

this will produce on output (if we have read our data from file ***mydata.tnt***) the equivalent of placing the contents of ***textfile*** within the QUOTE command, or:

quote Sorry. An error occurred. Cannot analyze file $dataset. ;

That is, it will produce as output:

Sorry. An error occurred. Cannot analyze file mydata.tnt.

Note that the replacement is literal. The use of @@ may help to write more readable scripts, but it is iffy and picky (it will

probably take you several tries to get it right).

RECURSION
Normally, the instructions within a file cannot invoke input redirection for the file itself. One exception is the use of **goto**, which can redirect input to specific parts of any file (including itself). Self-invokation can also be done, more simply, with the **recurse** command, which makes the file call itself (arguments given to **recurse** are read and transferred).

DIALOGS (WINDOWS ONLY)

The command ***opendlg*** allows defining functional dialogs, with ease. Several examples of dialogs are contained within the file with example scripts (zipdruns.exe).

TREE-BRANCH LEGENDS (reading/writing)

The command **ttag** handles overlapping legends (or "tags") in the branches of a "target" tree. A specific target (say, tree N) can be defined with **ttag * N**. Once the target is defined, a legend L can be copied onto branch B, with **ttag +R V;** (successive calls add text to the branch, concatenating). If you intend to start writing text from scratch, you may need to turn off tre-legend storing (with **ttag-;**), to eliminate preceding legends.

Once values have been written to a target tree, the value can be read/retrieved, with *$ttag R*.

In windows versions, when the symbol \ (back-slash) appears in some place of the tree-branch legend, the text following the slash is placed in a new line when the tree is displayed in the pre-view screen (or saved to a metafile). This can be used to place some legend above branches, and some legend below (say, bremer supports above, jacknifing below). The keys F11-F12 move the labels up/dn respectively. Also in windows versions, it is possible to display the tree-tags as colors (that is, the branch-colors corresponding to states 0-9), using **ttag:**.

READING AND EDITING TREES

The TREAD command reads trees in parenthetical notation. Although this is not a macro command, it is sometimes practical to use it in combination with special instructions (as in the ***hybtree.run*** routine, up in the scripts subdirectory of the TNT web page). The syntax of TREAD allows some shortcuts in the definition of trees: 1) a string followed by three periods (...) is equivalent to listing all the taxa whose names start with the string and have not yet been placed in the tree; 2) the expression ***@T N*** is equivalent to listing all the terminal taxa included in tree ***T***, node ***N***; 3) the expression ***+T N*** copies the subtree ***N*** of tree ***T*** (with its corresponding resolution; note this is equivalent to placing one terminal or node as belonging to the the

corresponding node); 4) the expression ***:name*** is taken to mean the list of all taxa containing the string "***name***".

It is also possible to effect changes to a tree using the EDIT command. EDIT X Y edits node Y of tree X; if the first argument (before X) is a closing bracket (**]**), then EDIT will not show the resulting tree on screen (this will normally be desirable in the case of scripts). After X and Y, specifying two numbers, J and K, causes node J to be moved as sister group of node K (if K is a descendant of J, then this reroots subtree J so that K is sister group to anything else in that subtree; if J is a terminal and it is not included in the tree, then J is added at the specified position). If instead of two numbers, a single one, preceded by /, is given (that is: /N), then node N of the tree is collapsed.

To eliminate taxa from a tree, pruning them, the PRUNTAX command can be used (followed by a list of the trees to prune, and then a list of the taxa or groups of taxa to prune, separated by a slash, /).

FORMATTED OUTPUT

The output can be formatted by making the QUOTE command not to include a carriage return (ASCII character 10) every time is executed. This is done by setting the quotes as "literal", with LQUOTE=. If conversion to ASCII characters is allowed (with LQUOTE[), then the symbol & inside a quote command (or inside the related command, **errmsg**), followed by a number N, will write on output ASCII character N. In non-windows versions only, character 0 (&0) is interpreted as erasing the last line written to stderr, and character 1 (&1) is interpreted as backspacing and erasing one character in stderr.

Although the number of decimals in the conversion of user variables can be set with the **macfloat** command, sometimes it is necessary to use a more specific format. This can be done placing a slash (/) after the single quote that precedes the user variable to be converted (say, "result"), as follows:

```
'/+-W.Dresult'
```

where W and D are numbers. The symbol + indicates that the sign (positive or negative) must be output always (default is writing sign only if negative), the - indicates that the string will be left-justified (default is that the string will be right justified), the value W indicates the total width (in characters) that the converted string will occupy, and .D indicates the number of decimals to use (default is the number set with **macfloat**).

In many cases, it may be desirable to prevent the program from producing its normal output (to the screen, or to the output file). This can be controlled with the SILENT command: SILENT = xxx makes xxx (console, file, buffer) mute, while SILENT - xxx makes it non-mute.

One additional trick:

```
loop 1 3
     QUOTE Opening file results#1.out ;
     LOG results#1.out ;
      . . .
     LOG/;
     stop
```

will produce as output:

```
Opening file results1out
Opening file results2out
Opening file results3out
```

This is so because the dot after the "1" is interpreted as a dot in a number with some decimals. Using a double period after the "1" the problem is solved:

```
loop 1 3
     QUOTE Opening file results#1..out ;
     LOG results#1..out ;
      . . .
     LOG/;
     stop
```

which produces the desired output:

```
Opening file results1.out
Opening file results2.out
Opening file results3.out
```

HANDLING STRINGS

It is possible to access only part of the strings (either pre-defined strings, such as taxon, character, or state names, or user-defined strings).

Usage is (for all examples, assume *string* contains "hypothetical"):

$string:N	displays the first N characters of "string" Examples: $string:5 displays "hypot" $string:20 displays "hypothetical "
$string:-N	displays the last N characters of "string" Examples: $string:-5 displays "tical" $string:-20 displays " hypothetical"
$string:+N	displays the characters of "string" following

the first N
Examples:
$string:+5 displays “hetical”
$string:+20 displays “” (nothing)

$string<X display characters of “string” preceding X
Example:
$string<t displays “hypo”
$string<c displays “hypotheti”

$string>X display characters of “string” following first occurrence of X
Example:
$string>t displays “hetical”
$string>c displays “al”

In the case of pre-defined strings which need specification of a number, the number must follow the format specification. Thus, if taxon number N is Xus_yus and you want to extract the generic and specific names only, correct usage is:

```
$taxon<_ N      generic name (“Xus”)
$taxon>_ N      specific name (“yus”)
```

The same applies to character or state names.

You can also compare strings using the Needleman-Wunsch algorithm, and calculate a similarity value, with the *stringsim* expression.

Finally, you can also find where in string a given substring is, with the *isinstring* expression:

```
isinstring [ hypothetical hyp ]       returns 3
isinstring [ hypothetical pot ]       returns 5
isinstring [ hypothetical nowhere ]   returns 0
```

Note that *isinstring* returns the position in main string where the secondary one *ends* (instead of where it *begins*, which would return a useless 0 for two identical strings!).

PARSING INPUT FILES

TNT also offers some rudimentary capabilities for file parsing, with the **hifile** command (for ***H***andle ***I***nput ***File***). You can open file XXX with **hifile open**:

hifile open XXX;

Subsequent to opening XXX, invocation of some expressions returns or displays values from XXX:

```
$hifstring      next string
hifchar         next character (reading it)
```

hifspy	next character (not reading it)
hifnumber	next number
hiflines	number of lines read

You can have up to 5 files open for parsing, and you can change, close, or list active files with:

hifile active
hifile close
hifile open
hifile list

You can move through the file, skipping parts of it, using options of the **hifile** command itself:

hifile skip [XXX] N	skip N characters
hifile skips [XXX] N	skip N strings
hifile seek [XXX] C N	read until finding character C, N times
hifile seeks [XXX] SSS N	read until finding string SSS, N times
hifile skipline [XXX] N	skip N lines (carriage returns)

In each of the cases, default N is always 1; if file XXX is not specified within square brackets, it will use the active (=default) one. When opening a new file, it automatically becomes active (if you want another one active, you have to change it explicitly).

KEY ADVICE FOR NON-PROGRAMMERS

A key advice for anyone trying to start developing scripts for TNT (or any other type of programming, actually): it is necessary to be very, very patient. Very. Very. And yet a little more. Nothing comes out right the first time. Never, ever. It is always necessary to try and try until things work like one wants. If one keeps trying, eventually things get to work.

On the other (more rewarding) side, once one is over the bump, progress can be made faster. Once a minimum of things have been learnt, using scripts makes running simulations or doing special calculations much easier, specially when compared to the difficulty of programming equivalent tasks using C or some standard programming language.