AB48.4.1

QUICK REFERENCE to B, by L.G.L.T. Meertens

Numbers are exact or approximate. You get an exact number even if you use *3.14* or *22/7*. You get an approximate number if you use *E* for the ten power, or if you use the ~ function (pronounced "about"). For example, *~1000 = E3*, and *~0.005 = 0.5E−2*. You may also write *~(a+b)* etc.
Warning: an approximate number is *never* equal to an exact number. If you want to test if you may divide by *x*, and if you are not very sure that *x* is exact, it is not safe to use the test *x <> 0* (which is shorthand for *(x < 0 OR x > 0)*). You should use *~x <> ~0*.
If functions like +, −, *, / and ** work on exact numbers, the result is also computed exactly, except if the exponent *n* in *x**n* is a fraction. (A formula like *a*x**2+b*x+c* stands for what is usually written as ax^2+bx+c: your computer cannot stand dancing lines and requires that you write * whenever you mean multiplication, even in cases like *2*x*.) Arithmetic on approximate numbers gives approximate results (which, for many purposes, are precise enough, and often are computed much faster). Functions like *root*, *sin* and *log* always give an approximate result. (So *root 4 <> 2* and *log 1 <> 0*). More details are given at the functions below.

Texts consist of characters and are written like *'Jack and Jill'* or *"Jack and Jill"*. (The characters meant are not Jack and Jill, but the "*J*", "*a*", etc. You may use any printing character and the space.) Which of the forms you use, the one with single quotes or the one with double quotes, makes no difference to your computer. Never confuse the number *747* with the text *'747'*. Whereas *747 = 3*249*, *'747'* is quite another text than the text *'3*249'*, and *'3'*'249'* is not even a text; to your computer it is meaningless. The number *747* can be used to do arithmetic; to your computer it does not consist of characters and it is written that way only because the dominant earthian species has twice five wriggly appendices sprouting from its upper tentacles and finds this clumsy notation convenient, and because you are (presumably) a member of that species and your computer tries to please you. The text *'747'*, on the other hand, cannot be used in arithmetic, and if you nevertheless try to do so, your computer will warn you. It really is three characters in a row. The so-called quotes on the outside do not really count. They only serve to make clear where the text begins and ends. If you say prayers, it does not mean that you say "prayers". But if you say "prayers", you don't say the quotes, do you? You can find out the length of a text with the function #. For example, *#'toe' = 3*. If you use *'* before and after your text, you can only use it inside if you double it thus: *''*. Your computer knows that you really mean it only once: *#'p''q' = 3*. The rules for *"* are similar.
But if you use the other quote sign inside than the one you use on the outside, you should not double it. So write either: *'He said: "don''t!"'* or: *"He said: ""don't!"""*.

Inside texts, you can use weirdos (which are known as conversions) of the form `e`. Your computer computes the value and replaces the conversion by a suitable text. For example, if *i = 239* and *j = 4649*, then *'`i` * `j` = `i*j`' = '239 * 4649 = 1111111'*. Within the conversions the need to double the outside quotes inside has disappeared: *'`#'toe'`' = '3'*. (Don't look too long at it if you don't want to strain your eyes.) On the other hand, if you use a single ` as character in a text, you have to double it.
You can join two texts thus: *'now'^'here' = 'nowhere'*, and you can repeat a text as many times as you want: *'ox'^^3 = 'ox'^'ox'^'ox' = 'oxoxox'* (just like *x**3 = x*x*x*). You can take texts apart thus: *'lamplight'@4 = 'plight'* (since the "*p*" is the fourth character) and *'scarface'|5 = 'scarf'*.
You may combine @ and |: *'Benedictine'@4|5 = 'edictine'|5 = 'edict'*, and *'Benedictine'|8@4 = 'Benedict'@4 = 'edict'*.
Forms with @ and | may be used as targets:
if *t* has as content *'Benedictine'*, and you tell your computer to

PUT 'zedr' IN t@4|5

it puts *'Benzedrine'* in *t*; if *t* is *'participle'* and you tell your computer to

PUT '' IN t|8@7

it puts *'particle'* in *t*; and if *t* is *'creation'* and you tell your computer to

PUT 'm' IN t@4|0

or to

PUT 'm' IN t|3@4

it puts *'cremation'* in *t*.

Compounds are a bunch of values grouped together. For example, if you want to keep track of which books you have lent when to whom of your friends, you may tell your computer to

PUT 'N&P', 'Mote' IN book
PUT 84, 3, 17 IN date
INSERT book, date, 'bearded gnome' IN books'lent

and your computer inserts *(('N&P', 'Mote'), (84, 3, 17), 'bearded gnome')* in the list of lent books it keeps for you. (Better ask him his name next time, though.)
You can obtain the fields (as they are called) by putting the compound in a compound target. In the example, your computer would obey

PUT book IN author, title

by putting *'N&P'* in *author* and *'Mote'* in *title.*
The following is a neat trick to swap the contents of two targets:

PUT a, b IN b, a.

This tells the computer to make the compound *(a, b)* and to decompose it into *(b, a).*

Lists are like lists you make to do shopping: if you and a friend of yours each make a list, and your list is

tooth paste
shampoo
cucumbers
yoghurt
muffins
birthday present for linda

and your friend has

birthday present for linda
shampoo
tooth paste
muffins
cucumbers
yoghurt

and you compare lists, you will exclaim: why, we have *exactly* the same list. Similarly, your computer considers {*t; s; c; y; m; b*} and {*b; s; t; m; c; y*} as the *same* list. In fact, it always sorts the entries in a list from low to high; if you tell your computer to

PUT {5; 7; 3; 2} IN a
INSERT 4 IN a
WRITE a

you will see {*2; 3; 4; 5; 7*} written. The same entry may occur several times in a list. If you tell your computer to

```
PUT {} IN letters
FOR c IN 'mississippi':
    INSERT c IN letters
WRITE letters
```

it writes back {*'i'; 'i'; 'i'; 'i'; 'm'; 'p'; 'p'; 's'; 's'; 's'; 's'*}.
You may insert all kinds of values in a list, but for each list they must all be the same type of value (all numbers, or all texts, etc.). You may use {*1..n*} as shorthand for {*1; 2; ... ; n−1; n*} and similarly {*'a'..'z'*}.

Tables are somewhat like dictionaries. A short English-Dutch dictionary (not sufficient to maintain a conversation) might be

aardvark:	aardvarken
apartheid:	apartheid
furlough:	verlof
of:	van
or:	of
van:	bestelwagen
yacht:	jacht

Table entries, like entries in a dictionary, consist of two parts. The first part is called the *key*, and the second the *associate*. All keys must be the same type of value, and similarly for associates. A table may be written thus: {*['I']: 1; ['V']: 5; ['X']: 10*}.
If this table has been put in a target *roman*, then *roman['X'] = 10*.
Your computer keeps the tables sorted by key. If you next tell your computer to

```
PUT 100 IN roman['C']
```

then *roman* will contain {*['C']: 100; ['I']: 1; ['V']: 5; ['X']: 10*}. You can find out what the keys are with the function *keys*; in the example, *keys roman* = {*'C'; 'I'; 'V'; 'X'*}.

PREDEFINED COMMANDS

HOW'TO c: commands
tells your computer how to execute *your* command c. It must not be used inside other commands.

YIELD f: commands
tells your computer what value it must yield for *your* formula f when it is computed. It must not be used inside other commands.

TEST p: commands
tells your computer whether *your* proposition p should succeed or fail when it is tested. It must not be used inside other commands.

CHECK test
checks if the test succeeds, in which case nothing happens, but aborts if the test fails.

WRITE e
writes the value of e on the screen. It gives new lines for any /-signs before and after e.

READ t *EG* e
asks an expression from you to put in t. The e tells your computer what type of expression to ask for (number, text, etc.).

PUT e *IN* t
puts the value of e in t.

DRAW t
draws a random number (from ~0 up to ~1) and puts it in t.

CHOOSE t *FROM* l
chooses at random an element from the text, list or table l and puts it in t. (The element is not removed from l.)

SET'RANDOM e
sets the random generator, using the value of e.

REMOVE e *FROM* l
removes the value of e from the list held in l. The value must occur in that list. It is removed only once.

INSERT e *IN* l
inserts the value of e in the list held in l.

DELETE t
deletes the target t. This is used mostly to delete entries from tables or to kill permanent targets.

QUIT
quits from a *HOW'TO* or refinement.

RETURN e
returns the value of e from a *YIELD* or refinement for further computation.

REPORT test
reports from a *TEST* or refinement whether the test succeeds or fails.

SUCCEED
reports success from a *TEST* or refinement.

FAIL
reports failure from a *TEST* or refinement.

IF test: commands
executes the commands if the test succeeds.

SELECT:
test: commands
.
.
.
test: commands
selects the first test to succeed and executes the commands after that test. At least one test must succeed. To make sure, the last test may be *ELSE*, which catches if all other tests fail.

WHILE test: commands
executes the commands if the test succeeds, and keeps repeating this while the test keeps succeeding. If it fails the very first time around, the commands are not executed at all.

FOR t *IN* e: commands
executes the commands for t ranging over the successive characters of e if e is a text, entries of e if e is a list, and associates of e if e is a table.

ALLOW t
allows the use of the permanent t inside a *HOW'TO*-, *YIELD*- or *TEST*-body. It must occur there at the head.

PREDEFINED FUNCTIONS AND PREDICATES

Functions on numbers

~x
returns an approximate number, as close as possible in arithmetic magnitude to *x*.

x+y
returns the sum of *x* and *y*. The result is exact if both operands are exact.

+x
returns the value of *x*.

x−y
returns the difference of *x* and *y*. The result is exact if both operands are exact.

−x
returns minus the value of *x*. The result is exact if the operand is exact.

*x*y*
returns the product of *x* and *y*. The result is exact if both operands are exact.

x/y
returns the quotient of *x* and *y*. The value of *y must not* be zero (i.e., *~y <> ~0*). The result is exact if both operands are exact.

*x**y*
returns *x* to the power *y*. The result is exact if *x* is exact and *y* is an integer. If *x* is negative (i.e., *~x < ~0*), *y must* be an integer or an exact number with an odd denominator. If *x* is zero, *y must not* be negative. If *y* is zero, the result is one (exact or approximate).

n root x
returns the same as *x**(1/n)*.

root x
returns the same as *2 root x*.

abs x
returns the absolute value of *x*. The result is exact if the operand is exact.

sign x
returns an exact number from {*−1..+1*} with the same sign as *x* (where, e.g., *sign ~0 = sign −~0 = 0*).

floor x
returns the largest integer not exceeding *x* in arithmetic magnitude (so, even if perhaps *3 > ~3*, *floor ~3* still returns *3*).

ceiling x
returns the same as *− floor −x*.

n round x
returns the same as *(10**−n)*floor(x*10**n+.5)*. For example *4 round pi = 3.1416*. The value of *n must* be an integer. It may be negative: *(−2) round 666 = 700*.

round x
returns the same as *0 round x*.

a mod n
returns the same as *a−n*floor(a/n)*. (Both operands may be approximate, and *n* may be negative, but not zero.)

*/*x*
returns the smallest positive integer *q* such that *q*x* is an integer. The value of *x must* be an exact number.

/x
returns the same integer as *(/*x)*x*. So, if *x* is exact, *x = (*/x)/(/*x)*.

pi
returns approximately *3.1415926535...* .

sin x
returns an approximate number by applying the sine function to *x*.

cos x
returns an approximate number by applying the cosine function to *x*.

tan x
returns the same as *(sin x) / (cos x)*.

x atan y
returns an approximate number *phi*, in the range from (about) $-pi$ to $+pi$, such that *x* is approximated by $r * cos\ phi$ and *y* by $r * sin\ phi$, where $r = root(x*x+y*y)$. The operands *must not* both be zero.

atan x
returns the same as *1 atan x*.

e
returns approximately *2.7182818284...* .

exp x
returns approximately the same as *e**x*.

log x
returns an approximate number by applying the natural logarithm function (with base *e*) to *x*. The value of *x must* be positive.

b log x
returns the same as *(log x) / (log b)*.

(There should also be a collection of simple matrix functions.)

Functions on texts

t^u
returns the text consisting of *t* and *u* joined. For example, *'now'^'here'* = *'nowhere'*.

t^^n
returns the text consisting of *n* copies of *t* joined together. For example, *'Fi! '^^3 = 'Fi! Fi! Fi! '*. The value of *n must* be an integer that is not negative.

x<<n
converts *x* to a text (see 5.1.2.2.b) and adds space characters to the right until the length is *n*. For example, *123<<6 = '123 '*. In no case is the text truncated; if *n* is too small, the likely effect is that your beautiful lay out is spoiled. The value of *n must* be an integer.

x><n
converts *x* to a text and adds space characters to the right and to the left, in turn, until the length is *n*. For example, *123><6 = ' 123 '*. In no case is the text truncated. The value of *n must* be an integer.

x>>n
converts *x* to a text and adds space characters to the left until the length is *n*. For example, *123>>6 = ' 123'*. In no case is the text truncated. The value of *n must* be an integer.

Functions and predicates on texts, lists and tables

keys t
requires a table as operand, and returns a list of all keys in the table. For example, *keys {[1]: 1; [4]: 2; [9]: 3} = {1; 4; 9}*.

#t
accepts texts, lists and tables. For a text operand, its length is returned, and for a list or table operand, the number of entries is returned (where duplicates in lists are counted).

e#t
accepts texts, lists and tables for the right operand.
For a text operand, the first operand *must* be a character, and the number of times the character occurs in the text is returned. For example, *'i'# 'mississippi' = 4*.
For a list operand, the number of entries is returned that is equal to the first operand (which *must* have the same type as the list entries.) For example, *3 # {1; 3; 3; 4} = 2*.
For a table operand, the number of *associates* is returned that is equal to the first operand (which *must* have the same type as the associates in the table.) For example, *3 # {[1]: 3; [2]: 4; [3]: 3} = 2*.

e in t
accepts texts, lists and tables for the right operand. It succeeds if *e#t > 0* succeeds.

e not'in t
is the same as *(NOT e in t)*.

min t
accepts texts, lists and tables. For a text operand, its smallest (in the ASCII order) character is returned, for a list operand, its smallest entry is returned, and for a table operand, its smallest *associate* is returned. For example, *min 'syrupy' = 'p'*, *min {1; 3; 3; 4} = 1*, and *min {[1]: 3; [2]: 4; [3]: 3} = 3*. The text, list or table *must not* be empty.

e min t
accepts texts, lists and tables for the right operand.
For a text operand, the first operand *must* be a character, and the smallest character in the text *exceeding* that character is returned. For example, *'i' min 'mississippi' = 'm'.*
For a list operand, the smallest entry is returned exceeding the first operand (which *must* have the same type as the list entries.) For example, *3 min {1; 3; 3; 4} = 4.*
For a table operand, the smallest associate is returned exceeding the first operand (which *must* have the same type as the associates in the table.) For example, *3 min {[1]: 3; [2]: 4; [3]: 3} = 4.*
There *must* be a character, list entry or table associate exceeding the first operand.

max t and *e max t*
are like *min*, except that they return the largest element, and in the dyadic case the largest element that is less than the first operand. For example, *'m' max 'mississippi' = 'i'.*

n th'of t
requires an integer in *{1..#t}* for the left operand, and accepts texts, lists and tables for the right operand. It returns the *n*'th character, list entry or associate. In fact, *n th'of t*, for a text *t*, is written as easily *t@n|1*. For a table, it is the same as *t[n th'of (keys t)]*, which is something different from *t[n]*, unless, of course, *keys t = {1..#t}*. For a list, *1 th'of t* is *min t*.