The GTL Programming Language

Programmer's Reference Manual

by

Doug Lennox

Lennox Computer

Edition 28 December 2023

The GTL Programming Language

Preface

GTL Stands for General Tuple Language. The language design and implementation is the work of Doug Lennox of Lennox Computer, and the design and implementation is the copyright property of Lennox Computer.

Some aspects of the design of the language are based on the work of Arthur Evans Jr. of MIT[1] and Martin Richards of the Computing Laboratory Cambridge[2], who in turn acknowledge the work or Peter J Landin of Imperial Collage London[3].

The purpose of the GTL language is to provide a very high level-programming environment for the future development of programming applications to be distributed and supported as part of Lennox Computer suite of business accounting and data-processing applications.

The major design criteria for GTL are as follows:

Logical Completeness	The applicative subset of GTL is a complete Lambda Calculus evaluation engine with fully implemented function objects bound to environments. The scope and extent of variables is comprehensively consistent with the rules of the Lambda Calculus. The L-value, R-value model is completely and consistently implemented with a thoroughly sound treatment of the assignment statement.
Notational Clarity	The declarative structure of the syntax and the handling of bound and free variables are modelled on traditional mathematical notation. e.g. **`let x, y = e1, e2 in f(x, y)`** or **`g(a, b, c) where a, b, c = e1, e2, e3`**
Simplicity of types	L-values are type-less. Basic R-values may be integers (32-bit signed), floating point (64-bit signed), strings, and tuples.
Fully Automatic Memory Management	The interpreter with built in automatic garbage collection administers memory allocation. The tuple is the primary construct available for building data-structures. Tuples are implemented with efficient operation as a primary goal.
32-bit Windows Environments	While GTL is an abstract, very high level-programming environment, platform independence is not an immediate goal of this project. The GTL compiler and interpreter are written in the C programming language with YACC-like support from a parser generator called PARGEN. The aim is to provide a sophisticated, high performance graphical user interface language specifically targeted at the Microsoft WIN32 environment, exploiting multi-threaded execution and the WIN32 virtual memory system. GTL is a production-programming tool.
Robustness	GTL is the antithesis of C. A meaningful error or exception at execution time will be accurately reported. Complete type checking will be implemented in all semantics and all library operators. It will be impossible for the interpreter to cause a memory addressing exception, or an illegal hardware operation.
Polymorphism prohibited	All library operators will accept only uni-morphic argument typing. (4)
Implicit type Conversions Limited	PAL above prohibited implicit type conversion completely. GTL will relax this ban only with respect to promotion of integers to floating-point. e.g. **`let i, x = 3, 3.414 in x + i`** will yield 6.414 without error. (5)

References

1. PAL Pedagogic Algorithmic Language, Dept Electrical Engineering, MIT, 1970.
2. The BCPL Programming Language – various papers and manuals, circa 1968 – 72
3. "The Next 700 Programming Languages", Landin 1966.
4. (Actually, this virtue is violated a fair bit to achieve practical API interfaces)
5. The introduction more recently of 64-bit values (R-values) means that is has been convenient to implement a variety of implicit type conversion between 32-bit, 64-bit and floating-point values – generally with promotion of a result to higher precision.

General Operation

GTL programs are created as ASCII text files with file type (extension) .gtl
e.g. TestCase.gtl

They may contain include statements which reference other .gtl files containing library functions or definitions.

A GTL program is compiled and interpretively executed by gtl.exe by passing the name of the file to be executed as the first parameter to **gtl.exe** – for example in a command prompt window one might type:

gtl TestCase.gtl

Of course, it is possible to associate the gtl extension with **gtl.exe** such that double clicking on a file with type .gtl will invoke the **gtl.exe** compiler/interpreter.

If the gtl program generates any output, a new window will be created and become visible to the operator containing the output. If it does not then the GTL program will execute invisibly as a windows "process" and can only be stopped by using windows Task Manager, if it does not exit naturally.

When the gtl execution window is open, there are two "threads" of execution present within a single window process – one executes the gtl program and the other implements the usual Windows message loop processing. When the gtl thread finishes, the windows message loop thread stays around to continue to handle mouse, and keyboard traffic to allow the operator to scroll and inspect the output generated by the completed execution. The operator may subsequently close the window when no longer interested in the program's output. If the operator attempts to close the window before the gtl execution thread is complete, then a warning dialog box is presented. If the operator chooses then to proceed with the close operation, then both threads are stopped and the window closed. This is a way of stopping a GTL execution which is out of control or no longer required.

In some situations, GTL programs may direct all of their output via a TCP/IP virtual circuit operating in either client or server role, and may therefore not create a visible window on the machine on which they are running.

Output by GTL Programs

GTL adheres to the original LISP concept that output is a natural consequence of expression evaluation. So where a GTL program consists of a single expression the output of the program is the value of that expression, and that output is shown in the default context which is initially the Microsoft Window associated with the GTL execution. The destination of the output may be changed (re-directed) by use of the **select** operator typically to a file or TCP socket. In That case there may be no visible Window, as GTL does not make the window visible until some output is directed to it.
Where a GTL program consists of a series of expressions separated by ';' characters then the values of these expression are output in sequence. For example:

```
let NL = "
" in
font("Arial", 10);
"This is the start of a GTL program execution"; NL;
"This is the 2nd line of output"; NL;
"This is the 3rd and last line"; NL
```

will generate three lines of output in the window, e.g.

This is the start of a GTL program execution
This is the 2nd line of output
This is the 3rd and last line

The GTL Lexical Scanner or Pre-processor

The Lexical Scanner is that part of the GTL Compiler/Interpreter which reads ASCII text file input, character by character and renders it into tokens or lexical items for further processing by the GTL Parser.
Some elements of the scanned input text are handled completely by the Lexical Scanner without being passed on to the Parser and these will be discussed here.

Form of Input	Action
Sequences of decimal digits not including a period terminated by a white space or punctuation character.	Passed on to Parser as a **ref** literal (signed 32-bit integer).
Sequences of decimal digits including a period terminated by a white space or punctuation character.	Passed on to the Parser as a **num** literal (signed 64-bit floating-point)
Sequences of alphanumeric characters starting with an alphabetic terminated by a white space or punctuation character. The underscore character is considered to be alphabetic.	Passed on to the Parser as a reference to an identifier, or a reserved word with the following exceptions.
Sequences of printable ASCII characters possibly including space and new-line, enclosed in a matching pair of double quotation marks. e.g.: `"The quick brown fox"`	Passed to the Parser as a literal string, which evaluates to a new string RV whenever the interpreter executes it. (Note: MS Word shows matching quotes thus " ", but in GTL the opening and closing quotation marks are the same character.)
Include statements e.g. `include "library.gtl"`	The Lexical Scanner opens the include file and continues processing input from there. Tokens for the include statement itself and its argument are not passed to the Parser. This means that include statements may be placed anywhere in a GTL source file and the compiler will parse the files as if they are joined together, and resume processing the outer file at the end of the included file. Include files may be nested to a maximum of 10 levels.
Define statements e.g. `define ITCustomer 1`	The lexical scanner reads the two arguments to the define statement, expecting an identifier and a literal respectively. The identifier is entered into the compiler's dictionary as a manifest constant with the value set to the second argument. A manifest constant is simply a name for a number. No memory is allocated. No L-value is created. Referring to it in a program is essentially the same as referring to a literal. Care should be exercised in the use of manifest constant defines, because the names defined do not obey the rules of scope and extent for variable names. Once a manifest constant is defined, because the pre-processor substitutes it, you cannot make a hole in the scope of the name by declaring it as a variable in nested definitions. Conventions such as all capitals, or a leading underscore character should be employed to keep the name spaces for manifests and variables separate.
Special defines:	**`define NOMENU 1`** inhibits the creation of the standard File menu and suppresses the Windows menu bar.

L-values & R-values in GTL

The model for L-values and R-values is based on the concept of a memory location which has an "address" (the L-value), and a "contents" the R-value.

The "L" and the "R" refer to left hand side and the right hand side of an assignment statement, where normal usage is that the left hand expression indicates the place to put something and the right hand expression indicates the new contents to put into that location.

The model used in GTL is more sophisticated and more abstract than the old BCPL and C usage, which was basically hardware oriented. The basic structure provides for two mappings between three sets.

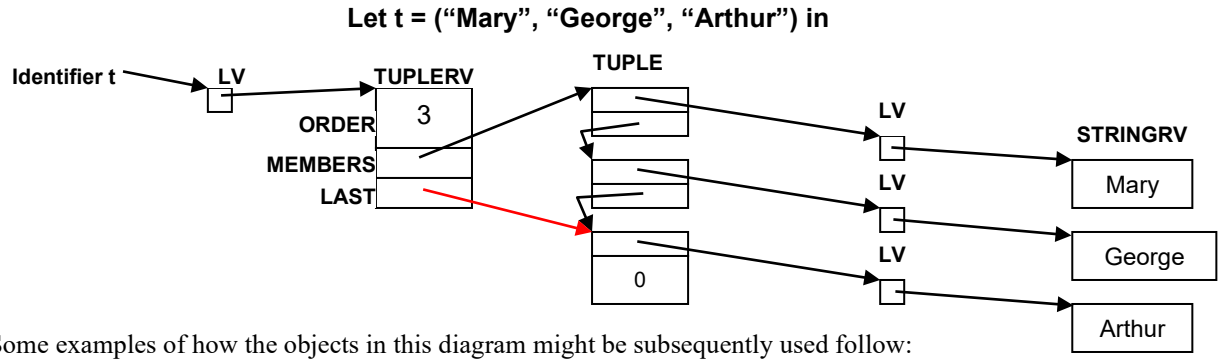
Identifiers → L-Values → R-Values

In a GTL declaration such as **let p = 3.414 in . . .** both mappings are initially established. The identifier p is mapped to a newly created L-value whose "contents" are initialised to the R-value 3.414. The mapping from the identifier to the L-value remains unbroken for the extent of the block at the head of which it is declared. However, the same identifier may be re-declared in an inner block making a "hole in the scope" of the outer declaration.

The mapping from the L-value to the R-value may be changed at any time by the execution of an assignment statement.

A tuple is a special kind of R-value, which contains an ordered set of L-values. The members of a tuple are "accessed" by applying the tuple R-value to an integer R-value to yield either an L-value or an R-value depending upon the context. Generally if a tuple is applied in the initialising expression of a declaration eg. **let x = t 4 in . . .** then the new identifier "shares" with the member of the tuple. That is to say, it maps to the same L-value so that if one is assigned to, the other also changes.

In the following example, the objects headed TUPLERV and TUPLE are a representation of the internal data structures used by the GTL compiler/interpreter in the implementation of tuples. They are not directly accessible by the GTL programmer.



Some examples of how the objects in this diagram might be subsequently used follow:

Example	Result
`let x = t 2 in`	The variable x maps to the L-value that maps to the R-value "Arthur". No new L-value or R-values are created.
`let x = $(t 2) in`	A new L-value is created - forced by the $ unshare operator and it is initialised to "Arthur".
`t 1 := "Henry"`	The contents of the 2nd L-value of the tuple are replaced by a new R-value.
`t := t aug "Harry"`	The identifier **t** maps to the same L-value, and it in turn maps to "the same" R-value, but that R-value has grown to a tuple of order 4 – ("Mary, "George", "Arthur", "Harry"). Any variables, which share with t, or any of the members of **t** will continue to do so (**aug** is theoretically inelegant but pragmatically efficient).
`let u = t au "Frederick"`	The **au** operator is a theoretically "pure" version of the **aug** operator and should be employed when perfect Lambda Calculus behaviour is desirable, typically in recursive applicable functions. **u** maps to a new L-value which does not share with **t**, but the first three L-values of **u** share with L-values of the members of **t**, and the 4th element of **u** comprises a new L-value and a new R-Value.

GTL R-Value Types

	Type	Attributes	Constructor	Description
1	**num**			This is a signed 64 bit floating-point number (double in C parlance)
2	**ref**			This is a signed 32 bit integer
	Int64			This is a signed 64-bit integer
3	**string**			This is a string.
4	**tuple**			A **tuple** is an ordered set of L-values, each of which may contain any type of GTL R-value. A tuple value may be applied to a **ref** value i to yield the i^{th} L-value (in L context) and the corresponding R-value (in R context). The operator *order* may be applied to a tuple value to yield a **ref** value indicating the number of elements. Tuples may be of any size the operator *aug* is available to add another element at the end of a tuple. Tuples may be created in GTL programs by means of the comma operator e.g. `let t = (x, y, z) in . . .`
5	**lambda**			A lambda expression with its free variables bound to an environment, ready for application in some other context.
6	**lv**			An internal only type used in the construction of tuples.
7	**dummy**			A useless type, returned by the assignment operator or another imperative style operator.
8	**primitive**			A built in function. Rarely used – most built in library operations are implemented as operators through the parser to avoid name lookup overheads at execution time.
9	**file**		input, output	A character stream I/O type. Used by the select operator, and the various input functions, lin, tin, stin, etc.
10	**coord**	x y	coord	Used to update the caret position in regards to the current caret position.
11	**absco**	x y	absco	Used to set the caret position.
12	**rect**	w h	rect	Used to draw a rectangle on the screen. The attribute w is the width of the rectangle and h is the height
13	**bitmap**	name x y	bitmap	Used to draw a bitmap on the screen. The attribute x is the width of bitmap to be drawn, and y is depth of the bitmap to be drawn. Name is the file name and directory of the picture to be displayed
14	**pen**	colour with style	pen	Used to set the colour of the pen. The pen is used for drawing the outline of rectangles on the screen. The colour attribute sets the colour of the pen, and the pen attribute is a handle to a Microsoft pen.
15	**brush**	colour style brush	brush	Used to set the colour of the brush. The brush is used for filling in the rectangle on the screen. The colour attribute sets the colour of the brush, the style attribute sets the style (either transparent of full), and the brush is a handle to the Microsoft brush.
16	**font**	name size	font	Used to set the font style and size. The attribute name is the font name. If the font name doesn't match exactly with one in system, the font is closely matched with one that does exist in the system. The colour is determined by the textcolour_rv
17	**textcolour**	r g b	textcolour	Used to set the text colour. r is the amount of red, g is the amount of g and b is the amount of b used in the final text colour.
18	**res**			A type used in the implementation of the valof/res semantics. The **res** type itself is not normally seen.
19	**date**			A 32-bit representation of a date, rendered as DD/MM/YY when output.
20	**time**			A 32-bit representation of a time, rendered as HH:MM when output. Held as seconds since midnight.
21	**field**			Used to allow an operator to input text.
22	**menu**	menu val	menu	Used to manipulate the menu in the oWindow. It contains a Microsoft handle to a menu item, and a return value
23	**line**			Used to draw a line on the screen using the current pen and brush and the current caret position.
24	**filemap**			A memory mapped file, accessible in virtual memory. Returned by the file operator, and available for transmission on a TCP/IP virtual circuit by the send operator.
25	**byte**		byte	A sequence of 8-bit bytes allocated in garbage collectable memory. The length operator may be applied to the **byte** value, and it may be applied to a ref in the range 0 to n-1 to access an individual byte member. The byte constructor does not initialise the values of the individual bytes.
26	**gif**		b2gif	A similar data structure to a **byte** type, but subject to interpretation as a GIF graphic image upon output to a device context.
27	**tga**		b2tga	A similar data structure to a **byte** type, but subject to interpretation as a TGA graphic image upon output to a device context.
28	**tiff**		b2tif	A similar data structure to a **byte** type, but subject to interpretation as a TIFF graphic image upon output to a device context.
29	**pict**		b2pic	A similar data structure to a **byte** type, but subject to interpretation as a PICT graphic image upon output to a device context.
30	**jpeg**		b2jpg	A similar data structure to a **byte** type, but subject to interpretation as a JPG graphic image upon output to a device context.
31	**png**		b2png	A similar data structure to a **byte** type, but subject to interpretation as a

				PNG graphic image upon output to a device context.
32	**wemf**		b2wmf	A similar data structure to a **byte** type, but subject to interpretation as a WMF graphic image upon output to a device context.
33	**pcx**		b2pcx	A similar data structure to a **byte** type, but subject to interpretation as a PCX graphic image upon output to a device context.
34	**pgm**		b2pgm	A similar data structure to a **byte** type, but subject to interpretation as a PGM graphic image upon output to a device context.
35	**bmp**		b2bmp	A similar data structure to a **byte** type, but subject to interpretation as a BMP graphic image upon output to a device context.
36	**abscale**		abscale	Used to set the absolute scale of an image
37	**scale**		scale	Used as a multiplication factor to modify the scale of an image
38	**eps**		b2eps	A similar data structure to a **byte** type, but subject to interpretation as a EPS graphic image upon output to a device context.
39	**child**			A child oWindow
40	**pbar**		progressbar	A progress bar
41	**btree**		bopen	A B* Tree
42	**semaphore**		semaphore	A semaphore
43	**pipe**			
44	**zip**		zipopen	A Zip file
45	**directx**		d3_window	A Direct X child window
46	**polygon**	((x0, y0), …	polygon	A 2 dimensional polygon value
47	**ellipse**	w, h	ellipse	A 2 dimensional ellipse value
48	**array**	n	array	A fixed size array of LVs which may be applied to a ref (0 to n-1)

Objects in GTL

GTL does not embrace the broad concepts of C language derived Object Oriented Programming languages as the philosophical direction o f GTL is towards advanced applicative concepts and the tuple, as the major data-structure paradigm.

However a form of object data-structure is implemented as an aid to data portability and crass polymorphism.

An object in GTL is represented by a tuple of pairs where the even elements are the property names which may be either integers (ref values in GTL parlance) or strings, and the odd elements are the values associated with pre-ceding property.

For example:

```
(pX, 1000, pY, 1000, pW, 2400, pH, 200, "Notes", "This is a graphic object example")
```

Where pX, Py etc are probably defined as ref literals.

Such objects are created and their properties area accessed using the @ operator in GTL. A property can be added or updated by using the @ operator on the left-hand side of an assignment statement. For example:

```
let g = () in

g @ pW := 4800;
g @ pX := 2000;
g @ pY := 3000;

etc
```

The use of the @ operator outside of a left-hand context yields the value of the property e.g. **g @ pX** evaluates to 2000.

If the property is not present in the object the value returned is $undef$.

The $undef$ value will be operates as a "unity" value for relevant operators - for example a string concatenated with $undef$ remains unchanged.

The Concept of Application in GTL

GTL contains a comprehensively implemented applicative evaluation interpreter based on lambda calculus principles.

The syntax used to indicate application is the juxtaposition of two expressions.

e.g

```
f x
```

Note that the traditional functional application notation from mathematics `f(x)` works, but according to the syntactic rules of GTL the brackets are redundant unless they are required to indicate precedence – for example:

f (x - 1)

versus

f x – 1

The semantic effect of application varies depending upon the types of the applicator and the applicand.

Applicator	Applicand	Operation
Lambda expression	Any type	The lambda expression is evaluated after binding its formal parameters to the value(s) of the applicand. If the applicand evaluates to a tuple, then the order of the formal parameters must equal the order of the tuple. However if there is a single formal parameter it will bind to any argument including a tuple. This is a powerful means of implementing variable or polymorphic parameterised functions.
Tuple	Ref	The application yields the i^{th} element of the tuple (0 <= i < n) where n is the order of the tuple.
String	Ref	The application yields the i^{th} character of the string.

String	String	The applicator is taken to be a regular expression and is applied to the applicand to yield a tuple of strings as a result of pattern matching the regular expression to the applicand string.
		
		Regular expression syntax:
		* matches any sequence of zero or more characters.
		? matches any one character.
		For example:
		
		`*<<*>>  "Dear <<Name>> our Mr <<SalesMan>> will be calling"`
		
		will yield the tuple
		
		`("Dear ", "Name", " our Mr ", "SalesMan", " will be calling")`

The GTL Operator Library

Operator **Argument** **Return Value** **Description**

String Operators

stem	**string**	**string**	Returns the first letter of the argument string as a string.
stern	**string**	**string**	Returns a string identical to the string argument excluding the initial character. Application to the null string will cause an execution time error.
	tuple	**tuple**	When applied to a tuple returns a new tuple whose members share with 2nd through the nth members of the argument tuple. Application to a null tuple returns a null tuple.
last	**string**	**string**	Returns the last character of the string as a 1 character string. Returns the null string if the argument is the null string.
	tuple	**tuple**	Returns the last element of a tuple as a tuple of order 1, the tuple element shares with the last element of the argument tuple. An execution time error is generated if () is passed to *last*
front	**string**	**string**	Returns a string comprising all but the last character of the argument string. Returns the null string if the argument is the null string.
	tuple	**tuple**	Returns a tuple comprising all but the last element of the argument tuple. The resulting tuple shares with the corresponding elements of the argument tuple. If () is passed to *front* an execution time error is generated.
length	**string**	**ref**	Returns the length of a **string, byte or shared memory** value. (may also be applied to a **file** value return by the input operator, and a **filemap** value returned by the file operator in these cases the size of the file in bytes is returned).
space	**ref**	**string**	Creates and returns a string of space characters the specified length.
textsize	**string**	**(w, h)**	The result is the size in twids of a rectangle which would enclose the argument string with reference to the device context of the video display. Useful for positioning the caret accurately with respect to output. The string may contain NL sequences, in which case the **h** element of the result is for a multi-line text rectangle.
textlength	**string**	**w**	Returns the length in twids of the string as it would display in the video device context. Only suitable for one line strings.
words	**string**	**tuple**	Breaks the string up at punctuation and white spaces. Punctuation is included in the result. Any control character except (CR, LF, TAB & SPACE) is considered punctuation and returned as a string of length 1.
whitesplit	**string**	**tuple**	Breaks the string up at white space. Returns a tuple of the strings that were separated by one or more white spaces. Sub-strings enclosed in quotation marks e.g. “The quick brown fox” are not split, but the quotation marks are removed.
whitesplitter	**string**	**tuple**	Similar to whitesplit above, except that quotation marks are not treated specially.
linesplit	**string**	**tuple**	Breaks the string at CRLF sequences, or LF sequences for unixy style strings, returns a tuple of strings with the CRLF or LF sequences eliminated, successive CRLF or LF sequences are indicated in the resulting tuple by null string members. In addition *linesplit* caters for the use of **(char 3),** control-C, used as a paragraph mark, and treats it similarly to an LF character. linesplit applied to a null string returns ().
split	**(string, string)**	**(string, string)**	e.g. `split("Suburb=Hamilton", "=")` returns (“Suburb”, “Hamilton”) split searches from left to right along its first argument, for a string equal to its second argument, and returns a tuple of two strings, being the substring to the left of the matching character, and the substring to

			the right of the matching character. If no match is found, split will return two null strings.
csvsplit	**string**	**tuple**	Given a one line CSV string, returns a tuple with the elements split on the commas – double quotes are optional delimiters - all elements are returned as strings.
subst	**(string, string, string)**	**String**	e.g. subst("/", "-", "TEST/1") returns "TEST-1" if args are **(c, s, t)** Every occurrence of string **c** in in string **t** is replaced by string **s**. Multi-character strings of any size may be passed as any of the arguments.
removeoccurence	**(string1, string2)**	**string**	Removes all occurrences of string2 from string1
upper	**string**	**string**	Converts the characters to their uppercase representations.
lower	**string**	**string**	Convert the characters of the string argument to all lower case as necessary.
•	**2 * string** **2 *tuples**	**string** **tuple**	The period operator is an infix operator, which takes two strings and concatenates them together. i.e. `"ab"."cd" = "abcd"` Either argument may also be a ref value in which case it is converted to a decimal string. Returns a new concatenated tuple whose members share with the members of the arguments.
cat	**tuple**	**string**	Efficient bulk string concatenation – given a tuple of strings returns a single string concatenated from all the argument strings.
catb	**tuple**	**byte**	Efficient bulk concatenation of **byte** data.
cats	**tuple**	**string**	Same as cat but with a single space between each string.
catl	**tuple**	**string**	Same as cat but with a NL (CRLF) between each string.
catc	**(t, s)**	**string**	Same as other cat operators but allows the separator string to be specified.
char	**ref**	**string**	Returns a string of length 1 of the character whose ASCII code is given as the operand. e.g. `char 27` creates a string of length one, containing the escape character.
ascii	**string**	**ref**	returns the ascii code of the 1st character of the string argument. Returns 0 for a zero-length string.
!	**S ! (a, b)**	**string** **byte**	The ! operator is an infix operator, which takes a string to its left and a 2-tuple to its right. It returns a substring of the 1st operand defined by an offset and size given as the 2nd operand. eg. `"Fred" ! (2, 2)` evaluates to "ed". If the substring overlaps the end of the operand string the result is padded with space characters. This is a convenient way of converting a variable length string to a fixed size. e.g. `s!(0, 12)` If **a** is >= the length of **S** the result is a string of **b** spaces. If the left hand argument to the ! operator is a byte or filemap value, then the value returned is of type byte, this is a convenient way of manipulating binary data, and extracting fixed size records therefrom. If the requested subfield overlaps the end of the argument, then the result is padded with null bytes.
blank	**string**	**ref (1/0)**	Determines if a string consists of white space only.
isnumeric	**string**	**ref(1/0)**	Returns 1 iff all the characters of the string are numeric digits. (0-9)
isalphanumeric	**string**	**ref(1/0)**	Returns 1 if at least one character of the string is in the range "0" to "9"
unspace	**string**	**string**	Removes all spaces from the string and appends them at the end so the result string is the same length.

#	**string**	**string**	Hash operator removes trailing spaces and control characters (including carriage return and line feed), from a string value. Converts <undef> to null string.
hts	**string**	**string**	Make a string HTML friendly – removes leading white space, trailing white space and converts multiple white space to a single space character.
keyform	**string**	**string**	Create s string to a plain form suitable for use as a key. All upper case, no punctuation.
alphaupper	**string**	**string**	Returns a string containing only upper-case alpha and numeric digits. No spaces.
xmltag	**string**	**string**	Converts human readable string into strict XML tag
ampup	**string**	**string**	Translates, <. >, &, ‘ , “ into **&** form.
unamp	**string**	**string**	Convert ampersand escapes of the form **&** to single character equivalents.
lookup	**string**	**lv/rv**	Returns an lv or rv depending upon context, where the string argument is interpreted as a variable name. If the string is undefined lookup will return nil.
manifest	**string**	**ref**	**define** identifiers are usually referenced directly in Gtl source, but sometimes it is desirable to be able to retrieve the value of a define with reference to a string value, and that is the purpose of the **manifest** operator.
charcount	**(s, c)**	**ref**	Counts the number of times a character appears in a string. Format of argument is (string, character to count)
atom	**string**	**ref**	*atom* returns a ref value that uniquely identifies the string. Repeated calls to *atom* for the identical string will yield the same integer. The first time (in a GTL execution) that atom is called 1 is returned, and for each subsequent call for a different string, the return value is incremented. This is intended to make the values returned by atom useful as tuple indices.
atomreset	**nil**	**dummy**	Resets the atom value to 1 for the current GTL program, and destroys all pre-existing atoms.
atomname	**ref**	**string**	*atomname* is the inverse of *atom.* It returns a string corresponding to the integer value originally returned by *atom*. If the ref value passed was not previously returned by atom, a GTL execution time error ensues.
urlcanonicalize	**(string, e)**	**string**	Invokes API function to canonicalize URL strings. If e is 1 the string is encoded, if e = 0 the string is decoded (i.e., %20 changed to space, etc).
md5	**string**	**string**	Returns a 32-character hex-decimal hash of the input string as defined by RSA in RFC 1320 – Security encryption applications.
rtf	**string**	**tuple**	Where the argument is a string of Rich Text formatted data, the result is a nested tuple decoded from the RTF data suitable for further interpretation.
segment	**file**	**tuple**	Decodes an UN/EDIFACT segment into a tuple form from an input file.
nextpart	**(Offset, Content, Boundary)**	**ref**	Offset is a ref should be 0 in 1st call, Content is a string of multipart MIME data, may contain binary bytes, and Boundary is string MIME boundary – return an Offset to the next MIME multi-part or 0 if there are no more.
embedhtml	**(s, u)**	**dummy**	If **u** = 0 **s** is a string of HTML, if **u** = 1 **s** is a URL string. The web browser is embedded in the currently selected window and the HTML is rendered as appropriate. Use `embed("", 0)` to un-embed.
spellcheck	**string**	**string**	Displays a spell checker dialog enabling the operator to make corrections and returns the corrected string.

Tuple Operators

null	**tuple**	**ref (1/0)**	Returns 1 when applied to nil, otherwise returns 0.
order	**tuple**	**ref**	Returns the order of the tuple.

	or **lambda**		When applied to a lambda object returns the number of bound variables.
member	**(s, t)**	**ref**	Where **s** is the string to look for in tuple t. If **s** is found in t, then `member` returns i+1 where i is the offset in the tuple of the string, else it returns 0. **s** may also be a ref, num or tuple – in the tuple case, a match is made on the identical r-value. The tuples are not compared structurally as in the case of comparison operators.
union	**(A, B)**	**tuple**	Returns the set union of two tuples A, B regraded as sets – i.e. if A and B have unique elements – so will the result. (Tip – `union(A, A)` is clever way of eliminating duplicate values from a tuple - i.e. making it a "proper" set). Members of the result share with members of the arguments.
join	**(A, B)**	**tuple**	Returns the set join (intersection) of A with B i.e. only these elements which are members of both arguments.). Members of the result share with members of the arguments.
reverse	**tuple**	**tuple**	Returns a new tuple whose elements are in reverse order. They share with the elements of the argument tuple.
memberprefix	**(s, t)**	**ref (1/0)**	Similar to member, except instead of requiring and exact match for the string, it returns the 1st offset which contains a string for which the 1st argument is a matching prefix. E.g. **`Electronic`** will match with **`Electronics`**
col	**(t, i)**	**tuple**	Returns a tuple formed from the ith column of a table (tuple of tuples). The resulting tuple elements share with the elements of the table tuple. The operator will fail if any element of the table is too short (order <= i)
sum	**t**	**num**	Accumulates the arithmetic sum of the tuple's elements and returns it as a single num value. Elements may be either ref or num types. Any other types will cause an execution time error.

has	**t has x**	**i+1**	*has* is an infix predicate which is true if the 2nd operand is a "member" of the tuple 1st operand, in the sense that it is an identical (sharing) l-value. e.g.
			
			`let x = "Hello" in`
			`let t = (x, "there") in t has x`
			will be true.
			
			The actual value returned is the offset in the tuple plus 1.

remove	**(t, n)**	**rv**	Removes the nth element from tuple t. n is a ref that is less than the value returned by the **order** operator. The return value is the rv that was removed from the tuple. EG
			
			`let NL = "`
			`" in`
			`let t = ("a", "b", 2, 3, 4, 5, 6, "c") in`
			`remove (t, 7); NL;`
			`t; NL`
			
			The output would look like:-
			
			`"c"`
			`("a", "b", 2, 3, 4, 5, 6)`
			
			Note: this is a fairly ugly operator. There is probably a better way of structuring your program without using it.
			
			One efficient role for the *remove* operator is in the implementation of a FIFO de-queue operation e.g.
			
			`if order Q do Next := remove(Q, 0)`

			Where elements have been added to the Q tuple with the aug operator.
copy	**tuple**	**tuple**	The copy operator makes an unshared copy of the argument tuple at the top level. Any nested tuples will still be shared.
t2a	**tuple**	**array**	Creates an array of L-Values sharing with the members of the tuple.

sort	**tuple**	**tuple**	Sorts the elements in a tuple in ascending order. If a member of the input tuple is a tuple, then it will sort on the first element of the tuple eg
			`sort(("abcdefg", 123456),("abcdefg", 12))`
			will result in an answer of
			`(("abcdefg",12),("abcdefg",123456))`
			As you can see from this example, if the first two elements of a tuple are identical then it will automatically scan down the tuple to find the next different element. If the a tuple has order 2, and another of order 3, where the first two elements in the tuple are the same, the order 2 tuple will be placed first. The sort function will only work if the elements are all of the same type. If the type is a tuple then the element types in the tuple must be the same as well. eg
			`sort((string, ref, num), (string, ref, num))`
			however sort will not work in the case of
			`sort((string, ref, num), (num, ref, string))`
			even though both are tuples, the first element type in each tuple differs.
			
			sort uses a very fast binary tree insertion algorithm. It will cope with large amounts of data efficiently. The only thing to beware of is that binary tree algorithms do not handle pre-sorted data efficiently.
			
			sort may (temporarily) use substantial amounts of virtual memory during its execution.

t2csv	**(tuple, tbs, ForEXCEL, decimals)**	**string**	Convert a one-dimensional tuple to a Comma Separated value string. Escape quote characters by doubling and adds surrounding quotes if string element contains a comma. **tbs** is a 0 or 1 value which specifies trailing blank suppression in string elements. **ForEXCEL** is a 0 or 1 value which specifies "EXCEL Friendly" behaviour, **decimals** is an optional one-dimensional tuple of integers to specify the number of decimal places required for the corresponding **num** element.
cryptor	**string** **tuple**	**string** **tuple**	**cryptor** reversibly encrypts the string value. **cryptor** reversibly encrypts all string values at the top level of the tuple and returns a tuple which shares with original except for encrypted elements.

Content Addressable Memory Operators

cam	**()**	**cam**	Returns and empty content addressable memory value.
cam	**(M, c, t)**	**dummy**	Adds an entry into a cam where c is a ref, or a string and t is any value.
cam	**(M, c)**	**dummy**	Deletes an entry with key c. Cams will remain extant while they remain in extent then they will be garbage-collected in the normal way. Cams are accessed by application to a key value (ref or string) and the application returns the value associated with string or **$undef$** if not found
cammap	**(M, L)**	**dummy**	M is a cam value and L is a lambda expression of the form fn(c, t) . . which is applied to each member of the cam in sorted order. Where F & T are string values defining a range of keys to be mapped.

	(M, L, F, T)		

Arithmetic Operators

abs	**ref** or **num**	**ref** or **num**	Returns the absolute value of argument type unchanged
sig	**(d, v)**	**ref (1 or 0)**	The predicate is true if argument **v** is significant to **d** decimal places.
rc	**num**	**num**	Rounds to the nearest cent. ("scientific" rounding)
cents	**num**	**num**	Rounds to the nearest cent. (truncated rounding)
sin	**ref** or **num**	**num**	Returns the calculated sine of the argument passed in radians
cos	**ref** or **num**	**num**	Returns the calculated cosine of the argument passed in radians
tan	**ref** or **num**	**num**	Returns the calculated tangent of the argument passed in radians
asin	**ref of num**	**num**	Arcsin function returns radians
acos	**ref or num**	**num**	Arccos function returns radians
atan	**ref** or **num**	**num**	Arc tangent in radians
atan2	**(y, x)**	**num**	Args may be a mix of ref and num - returns arctan(y/x) in radians.
urefcmp	**(a, b)**	**ref (1 or 0)**	Where **a** and **b** are both refs. Compares two refs ignoring their sign. If the unsigned value of **a** is less than the unsigned value of **b**, the return value is negative. If the unsigned value of **a** is greater than the unsigned value of **b**, the return value is positive. Otherwise, the return value is zero. (obsolete use `%<` `%<=` `%>` `%>=` instead)
round	**(num, ref)**	**num**	Rounds **num** to the decimal place specified by **ref**. ("scientific" rounding)

Predicates (Predicates are operators which return truth values, 0 or 1 in GTL)

istuple	**tuple***	**ref (1 or 0)**	Returns 1 if its argument is a tuple value.
isimage	*image**	**ref (1 or 0)**	Returns 1 if its argument is a image value.
isstring	**string***	**ref (1 or 0)**	Returns 1 if its argument is a string value.
isnum	**num***	**ref (1 or 0)**	True if the argument type is **num** (64-bit floating point).
isref	**ref***	**ref (1 or 0)**	True if the argument type is **ref** (32-bit signed integer).
isi64	**i64***	**ref (1 or 0)**	True if the argument type is a 64-bit integer
isdate	**date**	**ref (1 or 0)**	True if arg is a date
istime	**time**	**ref (1 or 0)**	True if arg is a time
islambda	**lambda***	**ref (1 or 0)**	True if the argument type is **lambda**
iswhitespace	**1 character string**	**ref (1 or 0)**	True if the character is a whitespace character
ischild	**child***	**ref (1 or 0)**	True if the argument is **child**.
issubstring	**(s, t)**	**ref (1 or 0)**	True if t is a substring of s. e.g. `issubstring("Old Farts", "Fart")` will return 1.
isundef	**$undef$**	**ref (1 or 0)**	True if the argument value is that returned by the @ operator when applied to an object which does not contain the requested property.
iscontrol	**control***	**ref (1 or 0)**	True if the argument type is a Windows control

Graphic or Screen Operators

Note: all screen (& printer) coordinates are expressed in "twids", where 1 twid = 1/1200 of an inch. The origin is the top left hand corner of the scrollable virtual presentation space (*VPS*), or the top left hand corner of the page 1 on a printer. The *x coordinate* increases from left to right and the *y coordinate* from top to bottom. Many printers have non-printable margins of about 300 twids.

Basic output in the GTL language is achieved (LISP like) by an expression standing alone. For example:

```
let S = "Hello World" in
S; NL
```

The default output destination is the current (scrollable) window. The term Virtual Presentation Space or VPS is used to indicate the entire output space that can be scrolled to, either vertically or horizontally and it can potentially be many thousands of pages in extent.

When a string value is output as in the example above, a suitable clipping rectangle is associated with it, to precisely envelope the string. Where alternate clipping is desired - for example when a string exceeds the width of a pre-defined field, the clip operator may be used to limit the amount of string displayed.

coord	**(x, y)**	**coord**	Moves the caret to a new position relative to the current position. If (x, y) are num values they are interpreted as mm, rather than twids.
absco	**(x, y)**	**absco**	Moves the caret to an absolute position. . If (x, y) are num values they are interpreted as mm, rather than twids.
clip	**(l, t, r, b)**	**dummy**	Defines a default clipping rectangle for subsequent string output. Revert to natural clipping with **clip(0,0,0,0)**
line	**(x, y)**	**line**	Creates an r-value which will cause a line to be displayed when output. (x, y) are the absolute coordinates of the end of the line. The line starts from the current position. The line will be drawn with the current pen setting. The coordinate position is not changed. If (x, y) are num values they are interpreted as mm, rather than twids.
rect	**(w, h)**	**rect**	Draws a rectangle on the screen at the current position. The arguments specify the width and height of the rectangle. If (x, y) are num values they are interpreted as mm, rather than twids.
roundrect	**(w, h)**	**roundrect**	Draws a rectangle with rounded corners (200 twid dia.) on the screen at the current position. The arguments specify the width and height of the rectangle. If (x, y) are num values they are interpreted as mm, rather than twids.
polygon	**(x_0, y_0), (x_1, y_1), . . .**	**polygon**	Creates a value which when output, draws a filled 2-dimensional polygon on the screen using the current brush and pen settings.
ellipse	**(x_w, y_w)**	**ellipse**	Creates a value which when output, draws a filled 2-dimensional ellipse on the screen using the current brush and pen settings. If (x, y) are num values they are interpreted as mm, rather than twids.
pointin	**((x, y), v)**	**0/1**	*pointin* is a predicate which returns 1 if the 2D point passed as its first argument lies within the object passed as the 2nd argument. The 2nd argument **v** may be a **rect**, **polygon**, or **ellipse** value. Note that **rect** and **ellipse** values are only well defined when preceded by an *absco* call.
boundingrectangle	**graphic r-value**	**4-tuple**	(left, top, width, height)
bitmap	**(f, x_w, y_w)**	**bitmap**	Displays a bitmap image on the screen. An order 3 tuple must be passed to the function containing the file name, the width and height of the image to be displayed. The bitmap will be stretched to fit the specified rectangle. If either **x_w, y_w** are specified as zero, then the aspect ratio of the image is preserved.
brush	**(R, G, B) or nil**	**brush**	The values passed to this function set the colour of the brush to be displayed on the screen. The tuple RV passed to this function must be in the format of (R, G, B), or if the parameter passed is **nil**, then the brush is transparent.
gradient	**(R, G, B) or nil**	**dummy**	Establishes a linear gradient target colour. Nil disables gradient fill.

horizontal	**0/1**	**dummy**	Establishes a direction for the gradient rectangles.
pen	**(R, G, B, W)** **or** **(R,G,B,W,S)**	**pen**	The values passed to this function set the colour of the pen to be displayed on the screen. The tuple RV passed to this function must be in the format of (R, G, B, W). i.e. Red, Green. Blue values in the range 0 to 255, and the width if the pen in twids. A width of zero results in a device dependent width of one pixel. The optional 5-argument form permits the pen style to be specified – for example 0 = PS_SOLID, and 5 = PS_NULL for a "transparent" pen.
setrop	**ref**	**ref**	This operator establishes the background mix mode for rect objects when they are painted in the VPS. The value of 13 is the default, and 7 is a useful value which XOR's the rect's brush colour with the background. The operator returns the previous mix mode.
textcolour	**3-tuple** **(R, G, B)**	**textcolour**	The values passed to this function set the colour of the text to be displayed on the screen. The tuple RV passed to this function must be in the format of (R, G, B)
choosecolor	**((R,G, B),C)**	**((R,G, B),C)** **or** **nil**	Displays a ChooseColor dialog box to the operator, allowing a standard or customer colour to be specified. The (R,G,B) argument is the initial colour selection for the dialog box, and C is a 0 to 16-tuple of custom colours. C may be nil if there are no pre-defined custom colours. The result returned has the selected colur in its (RGB) component, and the tuple of custom colours if any. Nil is returned if the operator cancels the dialog box.
menucolour	**()**	**(R, G, B)**	The 3-tuple returned specifies the current menu colour for the version of Windows in use, usually a shade of light gray.
syscolour	**ref**	**(r, g, b)**	Returns the windows system colour specified by the index passed to syscolour. Uses the GetSysColour API.

choosefont	**(s, p, (r,g,b))**	**(s, p, (rgb))**	Displays a ChooseFont dialog box to allow the operator to specify a font by *type face*, *point size* & *colour*. e,g,
			`let Face, Points, Colour =`
			`  choosefont("Arial Bold", 10, (255, 0, 0)) in`
			`{ font(Face, Points);`
			`  textcolour Colour;`
			`  "Example text output"`
			`}`
			The type face string is the name of the type face as known to Windows. This string may also have appended, any combination of the following attributes Bold, Italic, Underline, & Strikeout and these refer to the corresponding style selections and check boxes in the dialog box.
			
			Windows 7 has a bug which affects choosefont - use GTL library alternative Choosefont.gtl

messagebox	**(s, t, m)** **or** **(s, t, m, r)**	**0/1** **or** **ref**	Display a standard windows message box where the string **t** is the title of the box and the string **s** is its contents. Show buttons according to value passed as integer m. If the tuple is of order 3, or r = 0 then Wait for operator response, and return 0 if cancel clicked, and non-zero if another button is clicked. Otherwise, the return value is the value returned by the MessageBox API function. m = 0 -> OK button m = 1 -> OK and CANCEL buttons. m = 5 -> RETRY and CANCEL buttons
beep	**ref**	**dummy**	Does a MessageBeep
tone	**(ref, ref)**	**dummy**	Musical tone to audio output **beep(Frequency, Duration)** in Hz and mS.
ask	**(s, t)**	**0/1**	Displays an OK/CANCEL message box with t as the title and s as the content. Return 1 if OK clicked.
font	**(Face, Points)**	**font**	Sets the current font Type Face and size. The tuple RV passed to this function must be in the format (typeface, size) – e.g. ("Arial", 12)

	or **(Face, Points, Width)** or **(Face, Points, Width, Colour)**		The typeface string may also contain any of the attributes Bold, Italic, Underline & Strikeout as appendages. e.g. “Courier New Bold Italic”. Points argument may be **ref** or **num.** An optional 3rd argument permits the width of the font to expressed in twids. An Optional 4th argument permits the colour of the font's text to be expressed a a 3-tuple RGB value.
fontangle	**ref**	**font angle**	Sets the orientation of the text in 10ths of degrees. Specify 900 to rotate the text 90 degrees.
enumfonts	**()**	**tuple**	Returns a tuple of strings which are the names of all the TTF fonts available.
justify	**2 tuple**	**dummy**	Sets the justification. The first element in the tuple is the `LEFT` or `RIGHT` justification, and the second element in the tuple is `VERTICAL` or `HORIZONTAL` alignment. This justify command is to be used in conjunction with the coord or absco command. LEFT will align the text to the left at the current caret position. RIGHT align will position the text at the right of the caret. HORIZONTAL will increment the X position in the current window after the text that is displayed. VERTICAL will increment the Y position in the current window to be below the current text. The current font style and size determine this. For non-“server reporting” applications the `(LEFT, HORIZONTAL)` justification is the default setting. For server reporting the default is (0,0) as the server report process stores its own default settings and behaviour. The default behaviour in server report is to buffer all the text for a line before displaying it on the screen. Changing the values will result in server report displaying the text as it receives it. This default behaviour can be restored by using the `justify(0,0)` command.
getjustification	**()**	**2 tuple**	Returns the current setting.
clear	**()** or **(x_o, y_o, x_w, y_w)** or **(x_o, y_o)** or **c**	**dummy**	*clear* with () parameters, clears the entire *VPS* and all layers. With 4 parameters, it clears all the primitives contained within the area specified by 4 Tuple in the currently established layer (0 to 3). The 4 tuple format is (beginning x, beginning y, x width, y width). If any argument is of type num it is interpreted as mm. When 2 parameters are specified, the clear operator will delete only the topmost element whose top left coordinates are precisely at **(x_o, y_o)**. The rectangle occupied by that element will be invalidated, but no other elements will be deleted. The 2 parameters version will only delete an element from the current layer. When a single ref value is passed to *clear*, the operator deletes all primitives in the VPS with the specified *mousecode* or *capture* code.
cleari	**ditto**	**ditto**	Equivalent to *clear* except no invalidates are done – use *invalidate* operator to repaint window when ready.
extent	**()**	**(x, y)**	This operator returns a 2-tuple indicating the maximum extent in the x and y directions of graphics objects which have been output to the VPS. It forces a recalculation of the extents which is useful after the *clear* operator may have destroyed objects at the left or bottom of the VPS.
layer	**ref**	**dummy**	Establishes a layer number in the range 0 to 3 for subsequent text & graphical output to the window. The *clear* operator will only delete those output primitives which have been output with the same layer setting as the call to *clear* (where clear is called with parameters). The Print command in the file menu will only include output primitives in layers 0 & 1. Objects from layers 2 & 3 are for video display only.

indent	**ref**	**dummy**	Sets the automatic indentation of the left margin in the window.
page	**2-tuple** **(x_w, y_w)**	**dummy**	Sets the size of a page to the parameters (x width, y width). Before page is called the initial default page size is (9900,14025), which corresponds to A4 in portrait orientation. To use A4 in landscape invoke page(14025, 9900). Affects the currently selected window. Also important for print formatting (portrait vs landscape).
landscape	**0/1**	**dummy**	Force printing to landscape A4
getpage	**nil**	**(w, h)**	Returns the current page size setting of the currently selected window.
getselect	**nil**	**ref, file or child**	Returns a value for the currently *select*ed output destination it may be a **ref** value if a socket is being used, a **file** value if output is going to a file from the output operator, or a **child** window value if output is going to a child window. The value returned by *getselect* may subsequently be passed to *select* to restore the output destination.
getfont	**nil**	**tuple**	Gets the current font. The returned tuple can then be passed to the font operator. (return a 4-tuple (Face, Points, Width, Colour)).
gettextcolour	**nil**	**tuple**	Gets the current text colour. The returned tuple can then be passed to the textcolour operator.
getpen	**nil**	**tuple**	Gets the current pen. The returned tuple can then be passed to the pen operator. (r, g, b, w, s)
getbrush	**nil**	**tuple**	Gets the current brush. The returned tuple can then be passed to the brush operator.
getlayer	**nil**	**ref**	Returns a number in the range 0 to 3, being the currently established graphical output layer. See *layer* operator and *clear* operator.
curX	**nil**	**ref**	Returns the current X position of the caret
curY	**nil**	**ref**	Returns the current Y position of the caret
curI	**nil**	**ref**	Returns the current indent value in twids
curL	**nil**	**ref**	Returns the current line spacing for the current output font in twids plus the **leading** if any
transparent	**nil or 3 tuple**	**dummy**	If nil argument, then any transparency information in the image will be used, other wise specify a tuple of format (R, G, B) which indicates the colour to be shown as transparent.
intersectrect	**(R, T)**	**(x, y, w, h)**	Where **R** & **T** are of the form (x, y, w, h) - **intersectrect** returns 4-tuple.
leading	**ref**	**dummy**	The argument in twids is added to the basic line spacing of the font in force when a LF character is output to the VPS.
printwidthfactor	**num**	**dummy**	A scaling factor applied to the font width for a printer device context. E.g., **printwidthfactor 0.965**
strent	**(x, y)**	**string**	Arguments typically from getclick() Returns a string if the coordinates fall inside a VPS element. Null string if no match.
vps	**()** **(x, y, w, h)**	**tuple**	Returns a tuple of data extracted from the content of the Visual Presentation Space Full details of the tuple format returned are documented in the chapter entitled VPS Formats below. This form returns only those primitives encompassed by the rectangle so defined.
freeze	**(l, t, r, b)**	**dummy**	The argument in twids defines an area of the selected window which becomes unresponsive to mouse moves and clicks – global rubber band still works. Use freeze(0,0,0,0) to thaw.
graphic	**tuple**	**dummy**	Defines a tuple containing reactive graphical data – points lines rectangles ellipses and polygons, which are reactive to the mouse in real-time. (RG_POINT, (x, y)) (RG_LINE, (x_1, y_1), (x_2, y_2)) (RG_RECT, (x_1, y_1), (x_2, y_2)) It is expected that points defining the location of lines and other

			geometric shapes will often be shared tuples so that moving a point also move or re-shapes the parent graphic.

Database Operators (LILAC Database)

openlilac	**nil**	**dummy**	Opens the LILAC database
closelilac	**nil**	**dummy**	Closes the LILAC database
is_a34	**nil**	**ref**	1 -> 34-bit LILAC database in use. 0 -> 32-bit database
lock	**nil**	**dummy**	Locks access to the LILAC data base using a lock on the file lock.acc (or lock.a34). Consistent with action.exe interlocking. Serialised access to records like Company Data
unlock	**nil**	**dummy**	Releases database lock – always sandwich with lock.
setentity	**string**	**dummy**	Establish default Group/Company prefix for data base operations. e.g. `setentity "GPSY"`
find	**(rec, s)**	**(k, a)**	find expects two string arguments. The first argument **rec** should be the LILAC record type obtained from the first element of the DDS tuple, and the second argument **s** the database key as a string. e.g. `find(InvoiceDDS 0, "000001 ")` find returns a 2-tuple the first element **k** of which is a string indicating the full LILAC database key including prefix characters, and the second **a** is a ref address in the database of the corresponding record. The address element will be zero if the key is not found in the database. If the key is present more than once in the database (like key successor behaviour) then the key returned will be the **last** instance.
first	**(rec, s)**	**(k, a)**	first expects a two string arguments. The first argument should be the LILAC record type obtained from the first element of the DDS tuple, and the second argument the database key as a string. e.g. `first(InvoiceDDS 0, "000001 ")` first returns a 2-tuple the first element of which is a string indicating the full LILAC database key including prefix characters, and the second is a ref address in the database of the corresponding record. The address element will be zero if the key is not found in the database. If the key is present more than once in the database (like key successor behaviour) then the key returned will be the **first** instance.
row	**(dds, a)**	**n-tuple**	row reads a record (table row) from the LILAC database and returns it as a tuple. Argument one is the DDS tuple specifying the record format, and argument 2 is the address to read from.
nextrow	**(dds, a)**	**n-tuple**	Similar to row above, except that the record returned will be the next record of the specified type, found by means of a serial scan along data.acc from the address **a**. This permits audit trail style serial processing of Ledger Entries. Note, the record at **a** is not returned it is the next record if any. If no record is found by the end of file, nil is returned. **a** is modified to contain the new address. **a** is set to 0 at the end of file. Only records from the current setentity are returned unless the setentity is " " in which case records for all entities are returned.
like	**(k, a)**	**dummy**	Modifies the 2-tuple's 2nd element to be the address of the next like key in the b-tree of the same record type. If there are no more keys with same key value the 2nd element will be set to zero.
succ	**(k, a)**	**dummy**	Modifies the 2-tuple's 2nd element to be the address of the next key in the b-tree of the same record type. The 1st element may also change if the next key is not same as its predecessor.

succaddress	**Ref or**
	(ref, string)
		ref	Returns the address of the next record in the database. Example uses:-
			`a := succaddress(a);`
			`a := succaddress(a, leDDS 0);`

scan	**3-tuple**	**dummy**	The first is starting key string and the second is an ending key string. The third argument is a lambda expression (function) which will be applied to a two tuple for each key in the database found between the starting key and the ending key. The

			lambda expression is passed a 2-tuple argument each time containing the full database key string, and the address of the record.
scanb	**3-tuple**	**dummy**	Reverse direction version of scan - otherwise common code.
map	**4-tuple**	**dummy**	The first argument is a LILAC record type string. The second is starting key string and the third is an ending key string. The fourth argument is a lambda expression (function) which will be applied to a two tuple for each key in the database found between the starting key and the ending key. The lambda expression is passed a 2-tuple argument each time containing the full database key string, and the address of the record.
mapb	**4-tuple**	**dummy**	Reverse direction version of map - otherwise common code.
endmap	**nil**	**dummy**	When applied within a mapped or scanned lambda expression causes the most closely enclosing map to prematurely terminate.
inmap	**nil**	**ref**	Returns 0 if not currently in a map or scan, or else non zero.
put	**(dds, t)**	**dummy**	Creates a new row (record) in the data-base. The 1st argument is the record specification (DDS) tuple, and the 2nd argument is a tuple containing the fields to be written. The record is written at the end of data.acc and the key is inserted in the index in btree.acc. The key need not be unique if LKS behaviour is desired. To prevent LKS behaviour the application should check for pre-existence of the key prior to using put. put cannot be called in a map.
putr	**(dds, t)**	**ref**	Identical to put in all respects, except that it returns a ref value which is the database address of the newly created record. The database address is the byte offset, from the beginning of the file **data.acc,** of the first byte of the new record. putr cannot be called in a map.
delete	**nil**	**dummy**	delete() should be applied within the evaluation of the mapped function passed to map – it's effect is to delete the record (or row) passed to the mapped function during the execution of map. delete has no application in any other context.
deleteaddress	**a**	**dummy**	Where **a** is a ref value which is the address of a record in the lilac database. The record is deleted by writing zeros, and the btree key (if any) is also deleted.
putback	**(dds, t, a)**	**dummy**	Re-writes a record in the database at an address from which the record was originally read. If the record to be written is the same size as the record at the specified address, it is simply written back there. If it is smaller than the one being replaced it is padded with zero valued bytes, if it is larger than the original, the original is deleted, and the new record is written at the end of the file. The 3rd argument **a** is the database address to rewrite the record.
check	**(dds, t)**	**string**	To be used prior to put, putr or putback to check the integrity of the data to be put, with reference to the dds specification. Returns the null string for valid data and a non-null error message if an inconsistency is detected. Use when a crash is to be avoided such as in a server role, where the data comes from a remote client who may have an out-of-date dds picture.
addbtree	**(k, d)**	**dummy**	Adds the key and disk address to the btree.
newrecord	**(dds, t)**	**(k, d)**	Adds the new record to the end of data.acc and returns the database key and disk address to be used in a call to addbtree.
dbasepointer	**ref**	**dummy**	Writes (initialises) the LILAC Database Pointer, also re-computes the checksum in the DBStatus record at address 0.

actionsearch	**(soc, dds, k, da, count)**	**(t1, t2, t3, …)**	**soc** is a TCP/IP connection to **action** established using the connect operator. **dds** is a record specification (DDS) tuple. **k** is the database key as a string (must be 50 characters in length). **da** is the disk address of the record (maybe 0). **count** is the number of lines to be returned. The maximum number of records that can be returned can be determined by this algorithm:-
			
			`x := 8192 / (116 + sizeof(5 – no of keys in record))`
			
			`116 := 56 + 4 + 56`
			
			For example in a Order_Head we have:-
			The 2 key fields are the NameKey and the OrderNo

			X := 8192 / (116 + sizeof(5 – 2)) X := 8192 / (116 + 4 + 8 + 8) X := 60 If hyperlib.gtl is included, then one may call the GetActionSearchCount gtl function which takes a single argument which is the record specification (DDS) tuple. GetActionSearchCount(ohDDS); This can be used in the following context:- actionsearch(Action, ohDDS, " "!(0, 50), 0, GetActionSearchCount(ohDDS)); This operator returns a tuple of tuples. The format of the tuple is (key, display string, disk address)
actionread	**(soc, dds, da)**	**tuple**	Retrieves a record (table row) from the LILAC database via action and returns it as a tuple. soc is a TCP/IP connection to action established using the connect operator. dds is a record specification (DDS) tuple. da is the disk address of the record. Must not be zero.
actionnew	**(soc, dds, t)**	**da**	Creates a new row (record) in the data-base via Action. soc is a TCP/IP connection to action established using the connect operator. dds is a record specification (DDS) tuple. t is the tuple containing the fields to be written.
actionwrite	**(soc, dds, t, da)**	**dummy**	Re-writes a record in the database via action at an address from which the record was originally read. soc is a TCP/IP connection to action established using the connect operator. dds is a record specification (DDS) tuple. t is the tuple containing the fields to be written. da is the disk address of the record. Must not be zero.
actionfind	**(soc, t, k)**	**da**	soc is a TCP/IP connection to action established using the connect operator. t is a record type, typically the first element in the specification (DDS) tuple. k is the key to look for.
actionnext	**(soc, t, k, d)**	**da**	soc is a TCP/IP connection to action established using the connect operator. t is a record type, typically the first element in the specification (DDS) tuple. k is the key to look for. D is the disk address of the current key, can be zero.
actionprev	**(soc, t, k, d)**	**da**	soc is a TCP/IP connection to action established using the connect operator. t is a record type, typically the first element in the specification (DDS) tuple. k is the key to look for. D is the disk address of the current key, can be zero.
readbytes	**ref**	**byte**	A binary read operation from the LILAC **data.acc** file is performed. The argument is the **byte** offset in data.acc The number of bytes read and thus the size of the byte result is determined by the record size found in data.acc which is 16-bit field in the 1st two bytes of the record.
writebytes	**(b, a)**	**dummy**	(re)-writes a binary record **b** (of type **byte**) to the LILAC database which may have been read by readbytes above. If **a** is zero, the record is added at the end of **data.acc** and added to the btree. If **a** is non-zero then the record is written at that address, in which case it must be the right size for that record location. When **a** is zero, the Company Id field in the record is overwritten using the current **setentity** Id before writing!
readbad	**ref**	**byte**	Same as readbytes except always returns data even when error conditions exists. Use after a bad readbytes to get the damaged data.
readdata	**(a, n)**	**byte**	Read binary data from data.acc (or data.a34) n bytes from address a nor record structure – required for backup purposes. (a is i64 & n is i32)
datasize	**()**	**i64**	Returns the current file size of data.acc (or data.a34) as a 64-bit integer.

Semaphore Operators

semaphore	**(c, m, Name)**	**(s, e)**	Creates a semaphore RV. The first argument is the initial count of the semaphore. The second argument is the maximum count of the semaphore. The third argument is the name of the semaphore. The name is limited to 260 characters. Name comparison is case sensitive. If the name matches the name of an existing named

			semaphore object, this function requests SEMAPHORE_ALL_ACCESS access to the existing object. In this case, the Initial Count and Maximum Count parameters are ignored because they have already been set by the creating process. If the name is the empty string "", the semaphore object is created without a name. **e** is an error code – if non-zero **e** is code returned by GetLastError. If the semaphore has already been created by another process e will be **ERROR_ALREADY_EXISTS = 183** and s will be the semaphore.
pee	**semaphore**	**dummy**	The traditional Dijkstra pee function. If the semaphore is non-signalled, the thread will wait until the semaphore is signalled (i.e. the count is non zero).
vee	**semaphore**	**dummy**	The traditional Dijkstra vee function. This function increases the semaphore count by one.
peewait	**(semaphore, ref)**	**ref**	Same as the pee operator except that it will wait the specified number of milliseconds. The return value could be any one of the following:- -1: unknown error 0: the semaphore was signalled 1: the timeout interval elapsed 2: the thread that created the semaphore exited without signalling the semaphore.
closesemaphore	**semaphore**	**dummy**	Closes the handle to the semaphore. Once this function returns you can not call pee, peewait, vee, getsemaphorecount or closesemaphore.
getsemaphorecount	**semaphore**	**ref**	Returns the count of a semaphore. The count indicates weather or not the semaphore is signalled. The signalled state is not changed.
wakeup	**(semaphore, date, time, ref, ref)**	**Dummy**	Signals the semaphore when the system time of the computer matches the date and time pasted in. The operator then increments the "wakeup" time by the 2 refs. The first ref refers to days and the second ref refers to minutes. The following code demonstrates how to wake up at the 8am every day `Wakeup(sem, date(today()), s2t(08:00), 1, 0)`

File Operators

getsystemdrives	**()**	**tuple**	e.g. returns ((C:\, 3, V), (D:\, 5, V))
			Type 3 is a fixed drive.
			Type 5 is a CDROM drive.
			V may be volume information.
			Possible types:
			`#define DRIVE_UNKNOWN 0`
			`#define DRIVE_NO_ROOT_DIR 1`
			`#define DRIVE_REMOVABLE 2`
			`#define DRIVE_FIXED 3`
			`#define DRIVE_REMOTE 4`
			`#define DRIVE_CDROM 5`
			`#define DRIVE_RAMDISK 6`

getspecialpath	**ref**	**string**	let Desktop = getspecialpath CSIDL_DESKTOPDIRECTORY in etc – see Microsoft CSDIL definitions.

dirlist	**string**
	
	(string, date)
	
	(string, date, date)
		tuple	Returns a tuple of files based on the pattern passed in as the parameter. If a directory matches then a '\' is appended to the end of the directory name.
			`let f = dirlist("c:\*.*") in`
			`f`
			might display.
			`("winnt\", "autoexec.bat")`

dirdetail	**string**	**tuple**	Returns a tuple of tuples of the form: (Date, Time, FileName, Size) For each matching file of directory – Date & Time are of last write, and size in is in bytes.
servers	**()**	**tuple**	Returns a tuple of strings identifying the "servers" visible on the network.

shares	**string** or **()**	**tuple**	The *shares* operator returns a tuple of strings identifying Windows shared folders visible on a specified server. E.g.
			`shares "\\www";`
			returns
			`(IMail, Resene, IPC$, Inetpub, WebSites, Work, lilac3, LogFiles, ADMIN$, Data3, C$)`
			`shares()`
			returns the shared folders on the local computer.
			shares will return a string in the event of an error.

sharepath	**string**	**string**	Obtains the physical path on the current computer of a network share e.g. sharepath "Users" returns "c:\Users"
input	**string**	**file**	Opens a file so that data can be read in from it. The parameter is the file name. If an error occurs a string value is returned containing diagnostic information
output	**string**	**file**	Opens a file so that data can be written to it. If the file already exists the contents of the file are destroyed. If an error occurs a string value is returned containing diagnostic information
append	**string**	**file**	Opens a file so that data can be written to it. If the file already exists, it is opened and the file pointer is moved to the end of the file. If the file does not already exist it is created.
select	**file** or **child** or **ref** or **emfdc**	**dummy**	Selects the output file. Use parameter 0 to return output to the screen. If the argument is of type **child** output is switched to the *VPS* of a child window. If the argument is of type **ref** and is non-zero, output is sent to the Socket connection identified by the **ref** integer. If the argument is of type **emfdc** the wF2_EMFOutput windows flag is turned on.
pshsel	**child** **0** **file**	**dummy**	Use for selection of a child window for output, in preference to *select* above – match with a *popsel*() to return selection to an "outer" window. *pshsel* 0 (temporarily) selects the outermost window, and then *popsel* will revert to an inner window. Switch output from a window to a file.
popsel	**nil**	**dummy**	Must always sandwich with a *pshsel*.

flush	**file**	**dummy**	Writes all unwritten content to the file.
close	**file**	**dummy**	Closes a file stream.
tin	**file**	**tuple**	Return a tuple constructed from the next input line interpreted as a CSV file line.
stin	**file**	**tuple**	Return a tuple constructed from the next input line interpreted as a CSV file line. All values in the tuple are forced to be string RVs
lin	**file**	**string**	Return a string RV of the next ASCII line - strip CR chars if any, and remove LF from end. Multiple LF characters will be returned as null strings
bin	**(file, adr, length)**	**byte**	Binary record read from **file** at address **adr**, size **length**
token	**file**	**string**	Returns lexical token from input stream. (CSS Compatible)
htmltag	**file**	**tuple**	Parses the input file to find the next HTML tag delimited by <> characters. Returns (p, t, q0, q1, q2 , . . .) where p is any text preceding the tag, t is the tag, and the qn string are white space separated qualifiers.
xml	**file**	**tuple**	Parses data from the input file looking for a well formed xml entity. Returns a tuple representation of the XML data, nested as required. If errors are detected in the structure of the XML data a string is returned reporting the error. Skips any <?xml style tags by detecting the ? character.
attribs	**string**	**tuple**	Given a string of the form `aaaaaa="bbbbb" ccccc="ddddd"` returns an **object** (forgiving about unquoted values).
css	**string**	**tuple**	Given a string of the form `aaaaaa:"bbbbb"; ccccc:"ddddd"` returns an **object** (forgiving about unquoted values).
json	**string**	**object**	Return a GTL object (Name Value Pairs) with quotes removed from names & string values – possibly nested, by parsing JSON formatted data.
eof	**file**	**ref (1/0)**	Determines if the EOF has been reached.
filecopy	**(Destination, Source)**	**string**	Format of the tuple is (to file, from file). If the to file exists it is overwritten unless read only permissions are set on the file. Returns the null string on success, returns an error string from Lennox error module based on Microsoft error number to indicate failure.
movefile	**(From, To)**	**string**	Uses the API MoveFile function to "rename" a file or folder. Note order of arguments is reverse of filecopy. Returns error string (null string for success).
installfile	**(To, From)**	**string**	Copies a file and removes the read only permission if that is set. Does not overwrite newer files. Returns GLE string on error.
createtempfile	**(Path, NamePrefix)**	**string**	Returns a unique temporary file name.
create_dir	**string**	**ref**	Creates a directory. The return value determines if the directory could be created or not. If the return value is 0, then the directory was not created.
file	**string**	**filemap**	Creates a mapped view of the file in virtual memory, represented by a special RV of type filemap. The length operator may be applied to a filemap RV to return its length in bytes. The close operator should be applied to a filemap RV to deallocate its resources. If the file in question does not exist the operator returns a filemap value with a length attribute of zero.
invalid	**filemap**	**0/1**	Returns 1 if the handle value of the filemap is INVALID_HANDLE_VALUE or fails to Map file into virtual memory.
gle	**()**	**string**	Get Last Error from win32 API
filenametype	**string**	**tuple**	Given a file path return a 3-tuple e.g. (test.xlsx, xlsx, Microsoft

			Excel Worksheet)
getexticon	**(ext, 1/0)**	**icon**	Returns an icon for a file type 2nd arg specifies small icon.
loadicon	**(executable, i, Small)**	**icon**	Extracts an Icon from a executable file.
filesize	**string**	**int64**	Returns 64-bit file size - returns 0 if file non-existent.
filetime	**file or filemap**	**8-tuple**	After a file has been opened with **input** or **file** above, the **filetime** operator may be used to obtain its modification date and time. The tuple returned contains 8 ref's from MS SYSTEMTIME structure. **wYear -** Specifies the current year. **wMonth -** Specifies the current month; January = 1, February = 2, and so on. **wDayOfWeek -** Specifies the current day of the week; Sunday = 0, Monday = 1, and so on. **wDay -** Specifies the current day of the month. **wHour -** Specifies the current hour. **wMinute -** Specifies the current minute. **wSecond** - Specifies the current second. **wMilliseconds** -Specifies the current millisecond.
localfiletime	**file or filemap**	**8-tuple**	As filetime above except returns local time not UTC.
systemtime	**nil**	**8-tuple**	The result is similar tuple to that is returned by **filetime**, except thet the time in question is from the computer's clock in UTC.
localtime	**nil**	**8-tuple**	Local date time in same standard format
timezone	**nil**	**ref**	Returns the signed offset from UTC in minutes. AEST = 600
fileattributes	**string**	**ref**	Returns a 32-bit value from the Windows API GetFileAttributes function. `(fileattributes F) && FILE_ATTRIBUTE_DIRECTORY` determines if F is a directory. In the file does not exist *fileattributes* returns -1
getcurdir	**nil**	**string**	Returns the current working directory. The initial default working directory for a GTL program is the same folder containing the primary source file.
setcurdir	**string**	**string**	Changes the current working directory. Returns "" on success and an error message string on failure.
filedelete	**string**	**string**	Deletes a file specified by the parameter. If the delete succeeds or the file does not exist in the first place, the null string is returned. If there is an error then an error string is returned.
delete_dir	**string**	**ref**	Uses API RemoveDirectory - returns non-zero on success – use GLE to get error string.
save	**(string, byte)**	**string**	Writes a file on the disk using the binary image from the byte argument and the path indicated by the string. Returns the null string on success, and an error message string on failure.
savepdf	**(string, ref)**	**dummy**	Creates a PDF file from the present VPS. The string is the file name with path as required. The 2nd (scale) parameter is a percentage.
pdfopenfile	**string**	**(h, p, w, h)** or **string**	Opens a PDF file for direct access using the Quick PDF Library API. Returns a handle and the number of pages in the document, or an error string. W, h are the width & height of a page in numeric points values.
pdfclosefile	**ref**	**string**	The argument is a handle returned by *pdfopenstring*. Returns a null string on success and an error string on failure.
pdfmergefiles	**(string, string, string)**	**string**	Combines two PDF files into one. Argument one & two are the input file names & argument three is the output file name. Returns null string on success.
pdfextractpagetext	**(h, p, Opt)**	**string**	Returns all the text on the specified PDF page in a variety of possible formats depending on the Opt parameter. Opt 0 provides plain text with no coordinate data. Opt 3 is most useful
pdfpagecontent	**(h, p)**	**string**	Returns all the content on the specified PDF page in a fairly obscure encoding.

pdfunlock	**string**	**(p, w, h)**	Opens a PDF file and returns the number of pages and the width & height of a page in numeric points values
pdfout	**(p, dpi)**	**pdf**	Returns a **pdf** value which may be output to the VPS, **p** is a page number in the range returned by pdfunlock.
pdfgetpagetext	**(p, Opt)**	**string**	Same as pdfextractpagetext above for a pdfunlocked file.
gle	**nil**	**string**	Returns a string describing the GetLastError value from the Windows API.
dword	**(filemap, ref)**	**ref**	Returns a 32-bit value formed from the 4-bytes at the byte offset indicated by the second argument. The 1st argument may also be a **byte** or **sharedmemory** value
word	**(filemap, ref)**	**ref**	Returns an (unsigned) 16-bit value formed from the 2-bytes at the byte offset indicated by the second argument. The 1st argument may also be a **byte** value
netconnections	**nil**	**tuple**	Returns a tuple of 3-tuples, each of which identifies a file open on this server from the network ((FileId, LocalPath, User) , ...)
netfileclose	**ref**	**string**	Argument is a FileId obtained from netconnections above. Forces the file to close. Requires Administrator or Server Operator membership. Returns null string on success or error string on failure.
assignb	**(b, n, x)**	**dummy**	Update **byte** value b at offset n with 8-bit value x
assignw	**(b, n, x)**	**dummy**	Update **byte** value b at offset n with 16-bit value x
assignd	**(b, n, x)**	**dummy**	Update **byte** value b at offset n with 32-bit value x
extractpdf	**byte**	**(s, e)**	Attempts to find an embedded PDF doucmnet in the byte object by Seargcon for %PDF & %%EOF seqences. Use the ! operator to extact the PDF file from the byte object B!(s,e-s)

Enhanced Meta File (EMF) operators

openemf	**string**	**emf**	Opens a disk based EMF file. To play an EMF file, output the return value. Use **close** to close the EMF file.
emf_create	**string or nil**	**emfdc**	Creates a new disk or memory based EMF file. Returns the handle to the EMF's device context (HDC) for use with the select operator.
emf_close	**emfdc**	**emf**	Closes the EMF device context. No further output to the device context can occur once it is closed. If the emfdc is selected, you must call select 0 before closing the emfdc.
emf_getdimensions	**emf**	**(x, y)**	Returns the dimensions of a EMF.
rotateemf90	**emf**	**emf**	Rotate EMF 90 degrees – useful for portrait to landscape conversion
emf_copy	**(emf, s)**	**emf**	Create copy on disk with **s** as filename

Clipboard Operators

clipcopy	**string, tuple or bmp**	**dummy**	The argument is placed in the clipboard. If the argument is s string the clipboard will contain simple text which may be pasted by any windows application. If the argument is a tuple then the clipboard will contain an ETR which will typically only be sensibly interpreted by a GTL program.
clippaste	**nil**	**(type, data)**	Return the contents of the clipboard as a string, tuple or **bmp** or tuple of dropped files depending upon how it was copied there. **type** = 0 -> Empty Clipboard **type** = 1 -> String **type** = 2 -> Tuple **type** = 3 -> BMP **type** = 4 -> Dropped Files (**data** is a tuple of strings)
clipcopyfile	**string**	**dummy**	The argument is the full path to a file which is to be copied to the clipboard.

Menu Operators

menu	**2-tuple**	**menu**	Returns a menu RV. The 1st element in the tuple is used to determine the menu item name and location. Lennox Computer has established a standard menu-naming scheme. The location of the menu item is determined by the '_' in the string passed as the 1st element in the tuple `menu("Edit_Paste", 45)` will place a Paste menu item in the pop up menu Edit. If the pop up menu Edit does not exist, it will be created.
enable	**menu**	**dummy**	Enables a menu item.
disable	**menu**	**dummy**	Disables and greys a menu item.
wmcommand	**nil**	**ref**	(1) Returns the value of the menu item selected by the user. It also updates any fields displayed. (2) If **mouseactive** display objects are visible in the window, wmcommand returns the mouse code of any display object that is left mouse clicked. wmcommand 0 – returns a WM_COMMAND message from the outermost parent window.
wmtraffic	**nil** **ref**	**ref**	Returns the number of *wmcommand*(s) ready to be processed by wmcommand. If *wmtraffic* is non-zero the next call to *wmcommand* is guaranteed not to busy-wait. wmtraffic 0 – check for traffic in the outermost parent window.
removemenu	**ref**	**dummy**	Where the argument is the id of a menu item to remove from the menus

Input/Output Operators

kb	**nil** **0**	**string**	Waits for the user to press a key and returns a character value to the calling function. The values returned are standard ASCII mostly. If the window flag wF_AllKeyStrokes is set, then arrow keys return ANSI esc sequences. Control characters are returned naturally, except that shift-tab is translated as *char* 14. If zero is passed as a parameter the keyboard buffer of the outermost parent window is used,
shifted	**( )**	**ref**	1 -> the Shift key is down.
kbtraffic	**nil** **0**	**ref**	returns the number of characters in the keyboard input buffer 0 otherwise. If *kbtraffic* returns non-zero, *kb* is guaranteed not to busy-wait next time it is called. A zero parameter makes *kbtraffic* inspect the keyboard buffer of the outermost parent window.
peekkb	**( )**	**string**	Returns a 1 character string or a null string if kb buffer is empty.
peekesc	**nil**	**string**	Returns the 2 character escape sequence without removing them from the queue. This call will not block if there are not 3 characters in the keyboard queue, however the caller will receive garbage. Should be used after a call to kbesc().
kbesc	**nil**	**ref**	Returns 1 if there are at least 3 characters in the input buffer and the first is an ESC character, otherwise returns 0.

field	**6-tuple or 7-tuple**	**field**	**`field(title, lvalue, x, y, w, j [, c])`**
			Creates an R-value which when output, displays a field on the screen designed for text input. Regardless of the number of parameters used, the first 6 parameters for both field functions are the same. The first parameter is the title of the field, the second is a RV containing the initial text to be displayed in the field. The third and the fourth parameters define the top left-hand corner of the input field. The fifth parameter defines the length of the field, and the sixth parameter defines the justification inside the input field (LEFT or RIGHT).
			`let invoice = "000002" in`
			`field("Invoice Number: ", invoice, 2000, 1000, 4000, LEFT);`
			However the 7th parameter is the WM_COMMAND value to send when the user presses enter in that field. This only works on the last field on a page.

invalidate	**field** **()**	**dummy**	Causes the re-display of the data in a field after it has been changed by internal processing. With nil arg does an invalidate rect for entire client area.
setfieldfocus	**field**	**dummy**	Ensures that the caret & keyboard focus are in the specified field.
getfieldfocus	**()**	**field**	Returns the field value which has the current keyboard focus.
currentfield	**()**	**(l, t, r, b)**	Rectangle of current field
underlinetext	**0/1**	**dummy**	Text in fields to be underlined.
mouseactive	**ref**	**dummy**	If the argument is non-zero then subsequent output to the display window will be active to the mouse cursor. That is to say a dashed rectangle will appear around the output object when the mouse cursor is moved over it, and the **wmcommand** operator will return that code if the left mouse button is clicked over the output object. If the argument is zero subsequent output objects will not react to the mouse. This applies to any visible output in the window. When the left mouse button is down a red rectangle is shown to provide dynamic visible feed-back to the operator. Any object created with mouseactive behaviour should not straddle a page boundary.
hotcolour	**(r, g, b)**	**dummy**	Specify a colour for mouseactive objects when the mouse cursor is over them (a "hover" colour). The default hotcolour is a rich gold.
rcoffset	**ref**	**dummy**	e.g. `rcoffset 10000000` – the argument will be added to the mouseactive code returned via wmcommand to indicate a right click rather than a left click on the active object. The initial value before any call to rcoffset is 0
url	**string**	**dummy**	e.g. url "http://www.lennox.com.au" all objects created subsequently will be associated with the url until a url ""; is issued.
getclick	**()**	**(x, y)**	Returns the (twid) coordinates of the last mouse up (left or right click). Includes Scrolling but not zoom.
rs232c	**(C, B, P)**	**0/1**	Open Coms port for I/O e.g. `rs232c("COM1", 9600, "N")` to open comms port 1 at 9600 baud with no parity. Pairty may be specified as "N", "M", "E", "O". or "S". The return value indicates success or failure. Only one Coms port may be open at a time. There is a convention that upper case "COMn" turns on hardware handshaking and lower case "comn" does not.
serialin	**()**	**string**	Returns the data from the comms port input buffer – characters are returned as a single string from 0 to n characters in length. If there are no LF characters in the comms input buffer, serialin will return all the characters in the buffer as a single string, otherwise it will return a string comprising the characters up to and including the first LF character.
serialout	**string**	**dummy**	Outputs the string on the comms port.
serialflush	**()**	**dummy**	Data output by *serialout* is buffered in memory. Use *serialflush* to cause the actual transmission of the data. (Transmission is automatic upon the buffer becoming full, or *closecomms* being called).
serialtraffic	**()**	**ref**	returns the number of characters in the comms port input buffer

seriallines	**()**	**ref**	returns the number of LF characters in the comms port input buffer. If *seriallines* returns non-zero, *serialin* is guaranteed to return a string terminated by (the first) LF character in the comms input buffer.
closecomms	**()**	**dummy**	Shuts down a Com port opened by *rs232c* above, and frees resources. Only one Com port may be open at a time in a given GTL execution. So *closecomms* must be used to prior to a 2nd call to *rs232c.*
capture	**ref**	**dummy**	If the argument is non-zero any rectangles created (output) until a capture 0, will capture mouse input. That is to say the mouse cursor will change to an I-beam over the rectangle, and any left or right mouse clicks will be sent via the *mousein* operator.
rubberband	**ref**	**dummy**	Similar to *capture* above, except the mouse cursor shown in the affected rectangles will be a cross, and a rubber band rectangle will be created and dynamically changed while the left mouse button is held down. Rubber band rectangles will be left on the window until another left mouse down. In addition, *rubberband 0* may be used to turn off a rubber band rectangle.
mousein	**( )**	**(c, r, x, y, curs)** **r =** **0 left up,** **1 right up,** **2 left down,** **3 right down,**	If there is any mouse clicks captured a 5-tuple will be returned. **c** is the argument from *capture* or *rubberband* identifying the rectangle, **r** is 0 for a left click a 1 for a right click etc, **x, y** are coords in twids. If the mouse input fifo is empty, the execution thread will block until a mouse click. See *mousetraffic* below. If the wF_NotifyAllMouseMsg window flag is set then mouse traffic from anywhere in the client area will be returned, otherwise only mouse clicks from within captured rectangles will return data via *mousein.* The curs result indicates the type of mouse cursor presently being used: 0 : Not Sizing 1 : NS Sizing Double Arrow 2 : WE Sizing Double Arrow
anchor	**()**	**(x, y)**	Coordinates of the last Left Button Down event
mousetraffic	**( )**	**ref**	Returns 0 if the mouse input fifo is empty. If *mousetraffic* returns non-zero, *mousein* is guaranteed not to block.
mousemove	**ref**	**dummy**	Non-zero arg turns on real-time mouse move capture (not often necessary). When turned on wmcommand() returns four values (a, d, x, y) where a is the argument passed to mousemove, d is 1 if the left mouse button is down and x, y are the coordinates of the WM_MOUSEMOVE message.
mouseup	**ref**	**dummy**	Establishes value to be returned on left mouse up event.
mousewheel	**ref**	**dummy**	Non-zero arg turns on capture of mouse wheel activity in the currently selected window. *wmcommand*() returns this code followed by a signed multiple of 120.
mousestate	**( )**	**0/1**	Returns 1 if the left mouse button is down.
hourglass	**0/1**	**dummy**	*hourglass 1;* turns on the hourglass mouse cursor, *hourglass 0;* restores the previous cursor.
setcursor	**0/1**	**dummy**	*setcursor 1;* turns on the hand mouse cursor, *setcursor 0;* restores the previous cursor.
timer	**(c, t)**	**Dummy**	Creates a timer which returns a WM_COMMAND message with a value of c, (received via the wmcommand operator) every t milliseconds. Calling timer with t = 0 stops the timer.
getanyinput	**nil**	**(w, x)**	Returns a 2 tuple. Where w = 0, x = output from kb(). Where w = 1, x = output from mousein(). Where w = 2, x = output from wmcommand(). Where w = 3, x = 0, notification that a resize of the window has occurred.

print	**string** **(string, from, to)**	**dummy**	Where string is the name of a printer, send a print out-of the contents of the VPS to the printer in question without displaying a dialog box. For example: `print "hp LaserJet 1300 PCL 6";` *print* with the null string will display a dialog box. The 3 arg form allows a from & to page number to be specified as ref values.
copys	**ref**	**dummy**	Specify the number of collated copies to *print* using the printer driver's built-in multi-copy ability. Default is one.
printermargin	**(x,y)**	**dummy**	Specify a left-shift and an up-shift for the printed image in twids. Note GTL tries to implement nonprintable margins using data from the printer driver, but many (Asian) printer drivers don't do this very well so this operator provides a manual override.
printaspect	**(x, y)**	**dummy**	The ratio x:y determines the aspect of fonts when rendered on a printer. e.g. **`printaspect(17, 40);`**
duplex	**0,1,2 or 3**	**dummy**	If the printer supports double sided printing – use it 1 -> Long Edge Binding 2 -> Short Edge Binding 3 -> Simplex
rawprint	**(p, s)**	**string**	**p** identifies a printer, and **s** is the raw data to send to the printer. The whole document (label, etc) should be assembled in **s** before a single call to *rawprint*, as the function submits a job to the Windows print spooler.
getdefaultprinter	**()**	**string**	Returns the name of the windows default printer.
setdefaultprinter	**string**	**string**	Change the default Windows printer, result is null string if no error.
tooltip	**ref or tuple**	**dummy**	Displays a tooltip when the mouse is in the mouseactive or capture primitive rectangle. If just a ref is passed as an argument, then tooltips are disabled for this primitive. Otherwise the formats are:- (code, tip string) (code, tip string, title) (code, tip string, title, image) If either tip string or title is an empty string, then the tooltip will not display either the tip string or title. This is useful when wanting to show an image without either a tip string or title. Call tooltip 0 to remove all tooltips.
doubleckicktime	**ref**	**dummy**	Set the Double Click interval in mS

Time and Date Operators

time	**ref**	**time**	Converts a ref into time. The time is the number of seconds since midnight.
atodate	**ref**	**ref**	Converts a ref into ATO formatted time. The ATO formatted time is DDMMYYYY. The date is the number of days since January 1st 1900.
date	**ref**	**date**	Converts a ref into a date. The date is the number of days since January 1st 1900.
today	**nil**	**ref**	Returns the number of days since 1/1/1900 based on the system date. Use **`date(today())`** to get a date RV
clock	**nil**	**ref**	Returns the number of seconds since midnight from the computers' clock. Use **`time(clock())`** to obtain a time RV
milliseconds	**nil**	**ref**	Returns the number of milli-seconds since midnight from the computers' clock.
atodatecomp	**(a, b)**	**ref**	Compares two ATO date formats of the form 01072019 (refs) – returns -1, 0, 1

Network Operators (client context, server context, either)

ping	**(IP, mS)**	**ref or string**	Returns a ref value in mS for the ping response if any – failure or timeout return a string
computername	**IP**	**string**	Returns the fully qualified domain name of the computer at the specified IP address
connect	**(host, port)** **(host, port, tries)** **(host, port, tries, async)**	**ref** (or **string** if there is an error)	Client/Server connection to remote TCP/IP Server. Arguments are of the form ("203.34.177.3", "3001") i.e. IP address of client computer and TCP port that the server is listening on. *connect* returns a socket as a small integer which may subsequently be passed to *recv*, *send* or *select*. This is an interface to the WIN32 sockets version of Berkley sockets. Where DNS is in operation, the 1st element of the argument may be a fully qualified host name e.g. "lilac.lennox.com.au" If *connect* fails, it will return an error message string. If it succeeds it will return an integer identifying the socket. Use *close* to terminate the connection after the last response has been received. The optional 3rd argument is the number of time to re-try the connection. If it is omitted then there will be 10 tries. The optional 4th argument provides for asynchronous receive. It is passed via wmcommand() when data is available.
secure	**(s, m, h)**	**ssl**	Establish SSL/TLS encrypted connection. **s** is a socket number from connect above, and m = 0 returns certificate data, m = 1 returns an **ssl** value which may be used in place of a socket value for, *send*, *recvsocket* and *ready* operators. **h** is a string argument which is the host name of the server to establish Server Name Indication (SNI) when required – **h** may be the null string, if SNI is not required.
listen	**(h, p)**	**string – IP Address**	Establishes a TCP/IP Server listening on the port p, h is IP address as string e.g. "203.34.177.3" and p is port as ref e.g. 80 N.B. if h is the null string "", any address will be used. listen returns a string indicating the IP address upon which it is listening.
accept	**()**	**ref**	accept is optional - only required for clients which connect and then expect the server to be the first with data. If the client is the first with data then the server can go straight to recv.
certify	**(s, c, k)**	**ssl**	Server side SSL/TLS encryption – s is a socket from accept, c is certificate path and k is key path.
recv	**nil**	**3-tuple**	Receives data from a TCP/IP client, connecting to the port specified in the preceding listen . Result is of the form (S, Data, Good), where S is a ref identifying the socket, Data is a string RV of data received, and Good is ref Boolean value which is 0 if the client has disconnected.
recvsocket	**ref**	**string**	Receives data from a TCP/IP socket (client or server) and returns it as a string. GTL execution thread will busy wait (with sleep). Suitable for TELNET or SMTP type interaction. N.B. Returns null string when server closes connection.
recvbyte	**ref**	**byte**	Similar to recvsocket except handles binary data and returns a **byte.** Uses a 10 mega-byte buffer – suitable for handling large image, audio or video files e.g. in a HTTP client context.
recvline	**ref**	**string**	Similar to recvsocket, except that the input is guaranteed to be broken into individual strings for each line received. A line is terminated after a Line Feed character (char 10), or before an escape character (char 27). The Line Feed or escape characters are included in the string returned. This concept is necessary because on TCP/IP stream circuits, even when data is sent as individual lines, the use of the Nagle algorithm will coalesce these into single large message for network transmission.
recvblock	**nil**	**3-tuple**	Similar to recv above, except buffers characters till a carriage return is received, and returns strings starting with esc or control char.
ready	**ref**	**0/1**	When applied to a socket, returns 1 if there is data available, 0 if not. Or SOCKET_ERROR in the case of an error.

oktorecv	**nil**	**0 or Socket**	When a listen has been established, oktorecv will return 1 if data is ready from any client connection. Zero means idle.
oktoaccept	**nil**	**0/1**	When a listen has been established, oktoaccept will return 1 when accept is guaranteed not to block.
send	**(S, Data)**	**string** **filemap** **byte** **image**	Transmits data (back) to a TCP/IP Client. Argument is of the form (S, Data) where S is a ref identifying the socket, which must have been obtained from a preceding recv call, and Data is a string to transmit. *send* also works with a **filemap** value, and transmits the entire file - this is used by Lennox Computer's web server application for example. *send* returns the null string unless there is an error in which case it returns the error string. A **byte** value is sent as binary data.
sendasync	**(S, H, F)**	**dummy**	Similar to *send* above, but in addition creates a new thread for processing the send and returns in the current thread immediately so a web server may process further requests while a large file is being transmitted. H is a string of Header data (e.g. HTTP Headers), and F is a path for the file to send – useful for larger files where server responsiveness must not be compromised.
sendshared	**(S, G, N, B)**	**dummy**	Creates a shared memory object and runs a GTL program to provide data via the shared memory which is sent asynchronously to the socket by a background thread. S is the socket, G is the GTL program, N is the name is the shared memory and B is the buffer size.
passdata	**(N, D)**	**string**	Passes data D back to *sendshared* caller via shared memory N
recvpacket	**ref**	**string**	Receives data from to a TCP/IP Server. Argument is a small integer socket id returned by connect. The result is a string. It is assumed that the 1st four bytes received are the overall size of the data and they are removed.
sendpacket	**2-tuple**	**dummy**	Transmits data to a TCP/IP Server. Argument is of the form (S, Data) where S is a ref identifying the socket, which must have been obtained connect, and Data is a string to transmit. sendpacket computes the overall size of the data and perpends 4 bytes containing that integer. The Data may also be of type **byte** to permit the transmission of binary data.
recvchunk	**(s, n)**	**string**	Returns when precisely n bytes of data have been received by client on socket s – suits Transfer-Encoding: chunked situation.
etrrecv	**nil**	**(s, data, good)**	Similar to recv, except that Data is a tuple transmitted as an ETR. The data is in the form of a tuple unless an error occurs in which case the data is s string identifying the error.
etrsend	**(s, data)**	**dummy**	Similar to send, except that data is a tuple to be transmitted as an ETR
etrrecvpacket	**ref** **(ref, var)**	**tuple**	Similar to recvpacket, except the return value is a tuple which has been transmitted as an ETR. The result is in the form of a tuple unless an error occurs, in which case the result is a string identifying the error. The alternate arg form allows for a progressbar variable.
etrsendpacket	**(s, data)** **(s, data, var)**	**dummy**	similar to sendpacket except that data is a tuple to be transmitted as an ETR. Form with 3rd argument available to permit the specification of a progress variable.
shutdown	**ref**	**dummy**	Applied to a small integer S identifying a socket, causes WinSock to perform an orderly shutdown of the TCP virtual circuit. Typically used to disconnect a client after the server has sent everything in a stateless server context e.g. HTTP server. Note a HTTP client will disconnect eventually anyway, but this seems to cause resource leakage on the server.
ipof	**ref**	**string**	The argument should be the socket number returned by recv, the string returned is of the form "203.34.177.2" being the IP address of the client computer from which the data was received.

ip12of	**ref**	**string**	The argument should be the socket number returned by recv, the string returned is of the form “203034177002” being the IP adddress of the client computer from which the data was received, as a fixed length 12 digit string.
hostname	**nil**	**string**	Returns the fully qualified host name of the local computer as a string RV e.g. “www.lennox.com.au”
lookupdns	**string**	**string**	Argument is a fully qualified host name e.g. “lilac.lennox.com.au”, result is a IP address in string form e.g. “203.34.177.3”. The result may be passed as the 1st argument to listen. If the host name is not found in the DNS the result will be "0.0.0.0"
dnsquery	**(s, t)**	**ref** or **string**	Queries the DNS system - s is a string being queried typically a host name or a domain name. If t = 1 an A record is queried and a ref IP address returned. If t = 15 an MX record is queried and the hostname of the mail server returned.
dnsupdate	**(n, i, d, u, p)**	**string**	n is a fully qualified domain name, i is a binary IP address, d is a Microsoft domain, u is a user name and p is it’s password. Dynamic DNS update of the A record for the FQDN if access permitted. Credentials are for an appropriate (Administrator) on the DNS server.
macaddress	**ref**	**string**	Returns a string of the form 203.34.177.281 00-24-8C-48-24-56 That is to say an IP address and a MAC address separated by a space character. The argument allows handling of multiple adaptors **0** retrieves the 1st and **1** the 2nd and so on. If no adaptor is found the null string is returned.
adapters	**()**	**tuple**	Returns a tuple of tuples for each network adapter (Name, IP, DNS)

Diagnostic Operators

debug	**String**	**“”**	Presents a dialog box to the operator, indicating the GTL module name & line number as the title and the string argument as the body text. Presents OK & Cancel buttons. Clicking OK continues execution, Cancel terminates the process.
linenumber	**()**	**ref**	Returns the source file line number of the current GTL expression.
exeerr	**string**	**dummy**	Software Trap – allows gtl execution code to throw a fatal error dialog and terminate the process. If the program is in “Server mode” will throw an error to the client.
memorydata	**nil**	**tuple**	Returns a tuple of the form (“FreeCount”, 1234, “FreeBytes”, . ….) This is diagnostic information about the operation of the GTL automatic memory manager and garbage collector.
checkvar	**Variable name**	**3-tuple**	This diagnostic functions returns a 3-tuple containing the name of the variable as a string, the lv of the variable expressed as a memory address, and the present R-value contained by the l-value.

Memory Management Operators

byte	**ref**	**byte**	byte returns a value of type **byte** which contains the number of bytes specified by the argument allocated in garbage collectable memory which is initialised to zeros.
memcpy	**(b, s)**	**byte**	**b** is of type **byte**, and **s** may be of type **byte** or **string**. The returned value is a **byte** value concatenating the two arguments.
sharedmemory	**(Id, Size)**	**M**	Creates an object providing access to a named portion of shared memory – use ! operator to access bytes therein.
sharedexisting	**(Id, Size)**	**M**	Returns a pre-existing shared memory object, or an error string if there is none such. The Size parameter is a Maximum size. The length of the returned memory block will be a multiple of 4096 bytes as determined by the creator.
sharedn	**(M, i, v)**	**dummy**	Updates an 8 byte section of a shared memory object with the num value v.
sharedd	**(M, i, v)**	**dummy**	Updates an 4 byte section of a shared memory object with the ref value v.
sharedb	**(M, i, b)**	**dummy**	Updates a section of shared memory at offset I with byte data b.

Conversion Operators

b2s	**byte**	**string**	Converts a byte or a filemap value to a string value
b2n	**byte**	**num**	Converts an b byte value to a num (changes type)
n2b	**num**	**byte**	Returns an 8 byte value from the floating point argument.
s2b	**string**	**byte**	Converts a **string** value to a **byte** value (note - **string** may contain binary bytes such as 0 – uses allocated length not nul termination.)
s2r	**string**	**ref**	Assuming a string is composed of decimal digits returns the corresponding integer as a ref RV.
u2r	**string**	**ref**	Unsigned string to ref conversion – effective DWORD result.
r2s	**2-tuple** or **ref**	**string**	Converts an integer – ref RV to a string RV of length n where the argument tuple is of the form (x, n) where x is the ref value to convert and n is the desired string length. Where a single ref argument is given a string of length 0 to n is returned as required by the number of significant digits of the argument plus a leading minus sign for negative arguments.
r2sz	**2-tuple**	**string**	Converts an integer – ref RV to a string RV of length n where the argument tuple is of the form (x, n) where x is the ref value to convert and n is the desired string length, includes leading zeros in result.
r2b	**ref**	**byte**	Converts a ref value 32-bit, 4-byte byte value.
b2r	**byte**	**ref**	Converts the first 4 bytes of a byte to ref
n2r	**num**	**ref**	Converts a num RV to a ref RV.
r2n	**ref**	**num**	Converts a ref RV to a num RV
:	n:(w,d) n:d	**string**	The **colon** operator converts a num, ref or a 64-bit integer to a string, where w is the desired field width and d is the desired number of decimal places (typically 0 for integer types). Scientific rounding is used. (An alternative form `%:` is available for outputting ref values as unsigned 32-bit values) With a single ref parameter after the **colon** the result will be a variable length string with no leading space characters and the specified number of places after the decimal point.
d2s	**date**	**string**	Returns an 8 character string of the form YYYYMMDD e.g. 20010124 for the 24th January 2001.

date2string	**date**	**string**	Returns an 8-character string of the form DD/MM/YY
date2string4	**date**	**string**	Returns a 10-character string of the form DD/MM/YYYY
time2string	**time**	**string**	Returns an 8-character string of the form HH:MM:SS
s2t	**string**	**time**	Converts a string of form HH:MM or HH:MM:SS to the corresponding time RV.
s2tex	**string**	**time**	Converts a string to the corresponding time RV. More sophisticated text interpretation. Supports either 24 hour or 12 hour time, so an optional AM or PM can follow any of these date formats:- - HHMMSS or HH:MM:SS - HHMM or HH:MM - HH where MM and SS goes to zero - MM where the user has obviously mistyped HH (HH goes current hour) - User can also type in midnight or noon This function is more processor intensive than the s2t operator.
s2dex	**string**	**date**	Converts a string to the corresponding date RV. More sophisticated text interpretation. The operator supports the following formats: - - DDMMYY or DD/MM/YY assumes sensible century. - DDMMYYYY or DD/MM/YYYY - DDMM or DD/MM assumes this year. - DD assumes this year and month. - MMDD or MM/DD where the user has obviously mistyped DDMM - Also, text shortcuts such as today, tomorrow, yesterday
d2r	**date**	**ref**	Converts a date value to a ref value – i.e. days since 1/1/1900
s2d	**string**	**date**	Converts a string of the form DD/MM/YY or DD/MM/YYYY to the corresponding date RV
string2date	**string**	**date**	Converts a string of the form YYYYMMDD to a date value.
xmlstring2date	**string**	**date**	Converts a string of the form YYYY-MM-DD to a date value.
t2r	**time**	**ref**	Converts a time value to a ref value in seconds since midnight.
t2etr	**tuple***	**byte**	Converts a tuple into an ETR
etr2t	**byte** or **file**	**tuple***	Converts an ETR into a tuple, errors are reported by returning a string instead of a tuple.
s2n	**string**	**num**	Assuming a string is composed of decimal digits returns the corresponding double as a **num.**
f2b	**filemap**	**byte**	Creates a **byte** value the same size as the mapped file, and copies the contents of the file to the **byte** value.
tw2x	**ref**	**ref**	converts twids to video device coordinates according to video resolution. (horizontal)
tw2y	**ref**	**ref**	converts twids to video device coordinates according to video resolution. (verticall)
x2tw	**ref**	**ref**	converts video device coordinates to twids (horizontal)
y2tw	**ref**	**ref**	converts video device coordinates to twids (vertical)
hex	**ref**	**string**	8-bit argument is converted to 2 character hexadecimal string. E.g. 255 gives FF
hex2b	**string**	**byte**	Converts a hexadecimal string to a corresponding **byte** value.
i64	**ref/num**	**int64**	Coverts a numeric value to a 64-bit signed integer value
i32	**int64**	**ref**	Converts a 64-bit integer to a 32-bit
u2b	*	*	Converts **undef** to “”. Returns any other value unchanged.
hex2d	**string**	**ref**	Hexadecimal string to binary conversion

gulp	*	**dummy**	Evaluates and discards * any expression.
encodebase64	**string|byte|image|file map**	**string**	Returns an encoded string of printable characters as base 64 representation of the binary input.
decodebase64	**string**	**byte**	Converts a base 64 string back to the original binary form.
splitbase64	**string**	**string**	Inserts CRLF sequence every 74 characters into a base64 string for convenience of display/print.
shared2string	**sharedmemory**	**string**	Assumes a block of shared memory contains null terminated string data and returns a string value up to the 1st null terminator in the memory block which is typically a multiple of 4096 bytes in size.
format	**(Values, Sizes, Decimals)**	**string**	Return a single string spaced and formatted from the data values. Use with a fixed pitch font, as the **Sizes** tuple is in terms character counts.
formatrow	**(Values, Widths, Decimals)**	**row_rv**	An R-value which when output, paints a horizontal row with the values formatted to the column widths in twids, num and ref values are right justified, num value presented with the specified number of decimal places.

Image Conversion Operators

			Note: the operators of form b2xxx do not perform any conversion of the data. They just copy the bytes and set the type of the new rvalue accordingly. Image values are carried as binary images of the external file format (jpg, gif, png, bmp etc), and thus may be saved directly to files with appropriate extensions.
b2gif	**byte** or **filemap**	**gif**	Copies the data from the **byte** type into a **gif** type, such that if the **gif** type is output to a device context it will be rendered as a graphic image. **gif** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2tga	**byte** or **filemap**	**tga**	Copies the data from the **byte** type into a **tga** type, such that if the **tga** type is output to a device context it will be rendered as a graphic image. **tga** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2tif	**byte** or **filemap**	**tif**	Copies the data from the **byte** type into a **tif** type, such that if the **tif** type is output to a device context it will be rendered as a graphic image. **tif** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2pic	**byte** or **filemap**	**pict**	Copies the data from the **byte** type into a **pict** type, such that if the **pict** type is output to a device context it will be rendered as a graphic image. **pict** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2jpg	**byte** or **filemap**	**jpeg**	Copies the data from the **byte** type into a **jpeg** type, such that if the **jpeg** type is output to a device context it will be rendered as a graphic image. **jpeg** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2png	**byte** or **filemap**	**png**	Copies the data from the **byte** type into a **png** type, such that if the **png** type is output to a device context it will be rendered as a graphic image. **png** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2wmf	**byte** or **filemap**	**wmf**	Copies the data from the **byte** type into a **wmf** type, such that if the **wmf** type is output to a device context it will be rendered as a graphic image. **wmf** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2emf	**byte** or **filemap**	**emf**	Copies the data from the **byte** type into a **emf** type, such that if the **emf** type is output to a device context it will be rendered as a graphic image. **emf** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2pcx	**byte** or **filemap**	**pcx**	Copies the data from the **byte** type into a **pcx** type, such that if the **pcx** type is output to a device context it will be rendered as a graphic image. **pcx** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2pgm	**byte** or **filemap**	**pgm**	Copies the data from the **byte** type into a **pgm** type, such that if the **pgm**

			type is output to a device context it will be rendered as a graphic image. **pgm** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2bmp	**byte** or **filemap**	**bmp**	Copies the data from the **byte** type into a **bmp** type, such that if the **bmp** type is output to a device context it will be rendered as a graphic image. **bmp** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
b2eps	**byte** or **filemap**	**eps**	Copies the data from the **byte** type into a **eps** type, such that if the **eps** type is output to a device context it will be rendered as a graphic image. **eps** types are created in garbage collectable memory, so that they may be discarded at any time without penalty.
bmp2jpg	**bmp**	**jpg**	Converts a **bmp** to **jpg**.
jpg2bmp	**jpg**	**bmp**	Converts a **jpg** to **bmp –** returns string on error.
bmp2png	**bmp**	**png**	Converts (uncompressed) **bmp** to **png**
png2bmp	**png**	**bmp**	Converts **png** to **bmp** value (compressed)
image2bmp	*image*	**bmp**	Converts any image value to a **bmp**
image2byte	*image*	**byte**	Returns a **byte** type containing the image data for further manipulation.
rotate	(*Image*,D,BPP)	**bmp**	Takes any image in as its argument, and returns a bmp, rotated D degrees, BPP is the bits per pixel desired in the result.
getdimensions	*image*	**(w, h)**	Returns a 2-tuple specifying the width and height of the image argument in twids.
resizeimage	**(***image***, w, h)**	*image*	w & h are the desired width and height in twids. The image is resized using paintlib algorithms.
maketransparent	**(***image***, (r,g,b), T)**	*image*	For PNG images only, makes pixels of the indicated colour transparent in the returned *image* value. T is a ref value in the range 0 to 255 where 0 is fully transparent and 255 is fully opaque. The **(r,g,b)** is a threshold and those pixels with all 3 values >= the **(r,g,b)** argument will have the alpha channel set to the transparency value.
abscale	**num** or **(num, num)**	**abscale**	The value returned, when subsequently output to the VPS will affect the scaling of image values accordingly. 1.0 is 100%, 0.5 is 50% etc. If 2 arguments are passed the x and y scaling are separately specified.
scale	**num** or **(num, num)**	**scale**	Similar to abscale except when output the value passed is multiplied by the current scale factor to yield a new relative scale factor, which when output then affects subsequent image output.

Bitmap Operators

newbitmap	**(w, h)**	**bmp**	Creates a new image value of type **bmp** as a 24-bit bitmap initialised to all white.
setpixel	**(b, x, y, (r,g,b))**	**dummy**	Sets a single pixel in the bmp image b to the indicated RGB colour.
select	**bmp**	**dummy**	All output drawn to the bitmap until a **select 0** is called. In this case **select 0** returns the modified bitmap.

ETR Store Operators

etrput	**tuple**	**ref**	`etrput` writes a tuple as an ETR to the end of the **data.etr**, and determines the ETR Id from **admin.etr**, and appends the ETR Id and the ETR offset in **index.etr**. The ETR Id is returned.
etrputback	**(t, i)**	**dummy**	Where **t** is the tuple data to be put back into the store and **i** is the ETR Id. The ETR Id must pre-exist, i.e. have been returned by `etrput` at some time in the past, and not subsequently deleted. `etrputback` will try to write a tuple back to its current offset in **data.etr**. There are three (3) possibilities.

			The first is that the tuple is smaller than the previous tuple. The tuple is written back to the original offset, and the leftover space is freed, and this information about the leftover space is added to **free.etr**. The second is that the tuple is the same size as the previous tuple. The tuple is written back to the original offset, and nothing else needs to be done. The third is that the tuple is bigger than the previous tuple. In this case the tuple is written to the end of **data.etr**. The offset in **index.etr** is updated. The previous position of the tuple is freed, and this information is added to **free.etr**.
etrdelete	**ref**	**dummy**	`etrdelete` deletes the tuple from data.etr, removes the ETR Id from index.etr and the adds this information to free.etr. The ETR Id is then appended to the end of admin.etr
etrget	**ref**	**tuple**	`etrget` returns a tuple that matches the ETR Id else it returns nil. If the ETR Id has been deleted `etrget` returns nil.
openetrstore	**nil**	**dummy**	`openetrstore` opens the ETR Store for reading and writing. This must be called before any other operator to access the ETR Store. Any other calls made before this will generate a GTL execution time error. If there is no pre-existing ETR Store, empty files are created and initialised and the folder ETRStore is created if necessary.
closeetrstore	**nil**	**dummy**	`closeetrstore` closes the ETR Store. Any calls made (except `openetrstore`) after this will generate a GTL execution time error.
etrstoresizes	**nil**	**(D, I, F, A)**	Where D, I, F & A are respectively the size in bytes of the files which comprise the ETR store, viz: **data.etr**, **index.etr**, **free.etr** and **admin.etr**
etrnext	**ref**	**ref**	Returns the Id of the next sequential ETR from **index.etr** To find all Ids start from 0 and continue until zero returned.
etrsize	**ref**	**ref**	Given an ETR Id returns the size in bytes.
etrfreestats	**nil**	**tuple**	Returns a tuple of pairs (s_0, n_0, s_1, n_1, . . .) where the s_n are the sizes of free storage blocks, and the n_n are the counts of a given size.
etrcompress	**nil**	**dummy**	Makes **data.etr** a contiguous file - eliminating all free storage blocks, thereby reducing the size of **data.etr** to a minimum and the size of **free.etr** to zero.

BIN Store Operators 64-bit binary store

binput	**byte**	**ref**	`binput` writes **byte** data to the end of the **data.bin**, and determines the BIN Id from **admin.bin**, and appends the BIN Id and the BIN offset in **index.bin**. The BIN Id is returned.
binputback	**(b, i)**	**dummy**	Where **t** is the **byte** data to be put back into the store and **i** is the BIN Id. The BIN Id must pre-exist, i.e. have been returned by `binput` at some time in the past, and not subsequently deleted. `binputback` will try to write a **byte** data back to its current offset in **data.bin**. There are three (3) possibilities. The first is that the **byte** data is smaller than the previous tuple. The **byte** data is written back to the original offset, and the leftover space is freed, and this information about the leftover space is added to **free.bin**. The second is that the **byte** data is the same size as the previous **byte** data. The **byte** data is written back to the original offset, and nothing else needs to be done. The third is that the **byte** data is bigger than the previous **byte** data. In this case the **byte** data is written to the end of **data.bin**. The offset in **index.bin** is updated. The previous position of the **byte** data is freed, and this information is added to **free.bin**.
bindelete	**ref**	**dummy**	`bindelete` deletes the **byte** data from data.bin, removes the BIN Id

			from index.bin and the adds this information to free.bin. The BIN Id is then appended to the end of admin.bin
binget	**ref**	**byte**	`binget` returns a **byte** data that matches the BIN Id else it returns a zero length **byte**. If the BIN Id has been deleted `binget` returns zero length **byte**.
openbinstore	**nil** **string**	**dummy**	`openbinstore` opens the BIN Store for reading and writing. This must be called before any other operator to access the BIN Store. Any other calls made before this will generate a GTL execution time error. If there is no pre-existing BIN Store, empty files are created and initialised and the folder BINStore is created if necessary. If a string argument is supplied it will be used as the path for the BIN Store
closebinstore	**nil**	**dummy**	`closebinstore` closes the BIN Store. Any calls made (except `openbinstore`) after this will generate a GTL execution time error.
binstoresizes	**nil**	**(D, I, F, A, S)**	Where D, I, F, A, S are respectively the size in bytes of the files which comprise the BIN store, viz: **data.bin**, **index.bin**, **free.bin**, **admin.bin** and **sizes.bin**
binnext	**ref**	**ref**	Returns the Id of the next sequential BIN from **index.bin** To find all Ids start from 0 and continue until zero returned.
binsize	**ref**	**ref**	Given a BIN Id returns the size in bytes.
binfreestats	**nil**	**tuple**	Returns a tuple of pairs (s_0, n_0, s_1, n_1, . . .) where the s_n are the sizes of free storage blocks, and the n_n are the counts of a given size.
bincompress	**nil**	**dummy**	Makes **data.bin** a contiguous file - eliminating all free storage blocks, thereby reducing the size of **data.bin** to a minimum and the size of **free.bin** to zero.

The 64-bit binary store feature is available in the 64-bit & 32-bit editions of the GTL system to cater for potentally very large data-structures stored on the disk. It is generally similar to the **etrstore** concept with the notable exception that it does not assume the storage of etr data, although often etr data will be accomodated by using an expicit *t2etr* function with calls to *binput*, and *etr2t* with *binget*.

B* Tree Operators

bopen	**string**	**btree**	Where **file** is the name of the file containing the B* tree. bopen opens and selects the B* tree for reading and writing. `bopen` returns a **btree**. This must be called before any other operator to access the B* tree. Any other calls made before this will generate a GTL execution time error. If there is no pre-existing B* tree, an empty file is created and initialised.
bclose	**btree**	**dummy**	Where **b** is the **btree** returned by `bopen`. `bclose` closes the B* tree. If a B* Tree is selected, it is first deselected, and then closed. Any calls made (except `bopen`) after this will generate a GTL execution time error.
bselect	**btree**	**dummy**	Where **b** is the **btree** returned by `bopen`. `bselect` selects the B* tree for reading and writing.
badd	**(k, d)**	**dummy**	The first parameter is key string and the second is the data address. The key is inserted into the B* tree using the standard B* Tree algorithm. The value zero should not be used as a data address, because zero is used to represent a deleted key.
bdel	**(k, d)**	**dummy**	The first parameter is key string and the second is the data address. The key is deleted by using a "lazy" deletion algorithm.
bmap	**(k1, k2, f)**	**dummy**	The **k1** is starting key string and **k2** is an ending key string. The third parameter **f** is a lambda expression (function), which will be applied to a two tuple for each key in the database found between the starting key and the ending key. The lambda expression is passed a 2-tuple argument each time containing the database key string, and the data address of the record. The lambda expression should return a truth value (0 or 1) to indicate stop or continue where 1 -> continue with the mapping.
bmapb	**(k1, k2, f)**	**dummy**	Reverse direction version of *bmap*
bdiagmap	**()**	**dummy**	Creates a file `btreediagmap.txt` of b-tree diagnostics information.

B64 Tree Operators (B* Trees with full 64-bit capability)

b64open	**string**	**b64tree**	Argument is path for B64 Tree file. It will be created empty, if it does not pre-exist
b64add	**(b, k, i64)**	**dummy**	b is the **b64tree** value, k is the key string and **i64** is the 64-bit data. Duplicate keys are premitted and added chronoligically.
b64fwd	**(b, k1, k2, f)**	**dummy**	Calls lambda express f (s, a) for each key in the range (k1 to k2). f should return 0 or 1 to break or continue. s is a key string and a is the corresponding **i64** data.
b64bck	**(b, k1, k2, f)**	**dummy**	Calls lambda express f (s, a) for each key in the range (k2 to k1). f should return 0 or 1 to break or continue. s is a key string and a is the corresponding **i64** data.
b64del	**(b, k, i64)**	**dummy**	Delete the key k from the b64tree b. If the i64 value is 0 all like keys will be deleted, but if the i64 value is non-zero a single specific instance will be deleted.

SFTP – Secrure File Transfer Protocol Support

sftpauthenticate	**(u,pt,n,p)**	**string**	(URL, Port. UserName, Password) returns null string on succesful authetication – else non-blank error string.
sftpdirectory	**string**	**tuple**	Argument is path to directory e.g. "/order". Returns a tuple of pairs (FileName, Longentry)
sftpget	**string**	**byte/string**	Argument is path to file. Returns error string or byte data is sucessful
sftpput	**(string, byte)**	**string**	Writes file to remote folder identified by 1st argument. Return null string on success.
sftprename	**(From, To)**	**string**	Argements areboth paths. "Moves" a fileon remote server, non-blank string indicates error.
sftpend	**()**	**""**	Release resources. (sandwich with sftpauthenticate)

GTL Functions Associated with LILAC 3 Reporting

DDS	**string**	**tuple**	A field is defined by a tuple of information (Field Name, Field Type, Field Length). The string RV passed to DDS must be a valid Data Dictionary table name. The DDS function will then return the tuple of fields that defines the Data Dictionary table name. eg
			`DDS("Invoice_Total")`
			will return
			`("14", (ITInvoiceNo, 1, 8), (ITCustomer, 2, 12), (ITOrder, 2, 12), ...)`

NewRow	**DDS tuple**	**tuple**	NewRow returns a tuple which is isomorphic with the kind of tuple returned by the row operator from the data base. I.e. each element of the tuple is initialised to the correct type as indicated by the DDS tuple, using null strings and zero values as appropriate.

NameAddress	**string**	**tuple**	Returns the first row corresponding to the name and address key passed into the function.
			
			`let icar = NameAddress(it _ITCustomer) in`
			`\ where icar = Invoice Company Address Row`
			`\ and it is the Invoice Total record`

GetCreditor	**string**	**tuple**	Returns the first row corresponding to the creditor key passed into the function. There is built in support for Creditors in head office. Works in a similar way to the NameAddress function.
GetDebtor	**string**	**tuple**	Returns the first row corresponding to the debtor key passed into the function. There is built in support for Debtors in head office. Works in a similar way to the NameAddress function.

JulianDate	**ref**	**date**	Returns the Julian Meat Order Date.

Report	**3- tuple**	**dummy**	This function establishes the connection to the Lilac database, reads the company record, sets the entity, and then reads the company's street and mailing address. To use this function you must connect to through the L32 client, it will not work as a stand-alone function. The first element in the tuple is the title of the report to be displayed in the title bar of the window.
			
			The 2nd argument is the approximate total column width of the report in characters. The Report function will apply a suitable fixed pitch font to achieve optimal presentation and printability of the resulting report, based on this argument.
			
			`Report("Invoice Replication Report", 96, nil)`
			
			The 3rd argument may be a 2-tuple of of tuples, the members of which are the column titles for the report, with a string per column, assuming a 2-line column title style. For example:
			
			`(("Customer ", "Account ", " YTD "),`
			`(" Key ", " Title ", " Balance "))`

System Operators

command	**nil**	**string**	The command line with which the GTL compiler/interpreter was invoked. (As returned by the GetCommandLine windows API call).
envvar	**string**	**string**	Returns a string obtained from the Windows API call GetEnvironmentVariable. *envvar* will return the null string if the argument string is undefined.
setenvvar	**tuple**	**Dummy**	Sets the environment variable. First parameter is the environment variable name, the second parameter is the evironment variable parameter.
terminate	**nil**	**nil**	Terminates the current process and closes the GTL window immediately.
processid	**()**	**ref**	Returns Process Id of current process.
enumprocesses	**nil**	**n-tuple**	Returns a tuple of Process Id's for all the processes running on a system - c.f. Task Manager
processinfo	**ref**	**tuple**	Given a process id argument, this operator returns data about the process. If **t** is the result, **t 0** is a string giving the full path and filename of the base executable module. Some system processes return zero length strings.
processtimes	**ref**	**tuple**	Given a process id argument, this operator returns a 4-tuple of 64-bit nano-second values, (Creation Time, Exit Time, Kernel Time, User Time)
StackDump	**nil**	**dummy**	Diagnostic tool.
gettype	**any**	**string**	Returns a string containing the type of the RV to which this operator is applied.
username	**()**	**2-tuple**	For example (doug, CN=Doug Lennox,CN=Users,DC=lennox,DC=com,DC=au)
ldap	**4-tuple**	**tuple**	Example argument; "domain.lennox.com.au", "DC=lennox,DC=com,DC=au", "doug", "vulcan" That is Domain-Controller, Domain, User, Password.
win64	**()**	**0/1**	Returns 1 if the gtl interpreter is a 64-bit build. (g64.exe)
isremotesession	**()**	**0/1**	Returns 1 if running in a Remote Desktop or Terminal Services session.

Window Manipulation Operators

vscroll	**ref (1/0)**	**dummy**	Removes or displays the vertical scroll bar. By default the scroll bar is on. 0 removes the scroll bar and 1 displays it If vertical scrolling is on the up/down arrow keys are interpreted as line up/down scroll instructions. If vertical scrolling is off the up/down arrow keys are passed as keyboard input esc[A and esc[B The mouse wheel is disabled if vscroll is off.
hscroll	**ref (1/0)**	**dummy**	Removes or displays the horizontal scroll bar. By default the scroll bar is on. 0 removes the scroll bar and 1 displays it
scroll	**()**	**(h, v)**	Returns the horizontal and vertical scroll offset in twids (2-tuple of refs).
scrollwindow	**(x, y)**	**dummy**	Changes the scrolling of the current window by the signed x & y values.
scrollevent	**ref**	**dummy**	Allows the programmer to specify a value to be returned by *wmcommand()* to signal a change in the windows scrolling.
set_title	**string**	**dummy**	Sets the window title.
get_title	**() or ref**	**string**	The title bar contents are returned – nil arg specifies current window, ref arg specifies window handle.
click	**ref (1/0)**	**dummy**	Enables or disables left mouse button clicking in the window. The default is left mouse button clicking is enabled. 0 disables left mouse button clicking, and 1 enables it. If there are fields displayed in the window, the user will still be able to click in them, even if left mouse button clicking has been disabled.
setbgcolour	**3 tuple**	**dummy**	Sets the background colour of the window. The tuple is the format (R, G, B).
setresize	**3 tuple**	**dummy**	Sets the resizing options of the GTL Window. The first parameter determines if the minimize button is available, the second parameter determines if the maximize button is available, and the third determines if the border can be resized. In all cases if the value passed is 0, then that option is disabled, else it is enabled. `setresize(1, 0, 0)` Sets the window so that it has a minimize button but cannot be maximized or resized
showsysicon	**tuple**	**dummy**	Displays the default windows icon in the task notification area. The 1st argument is the text to be displayed when the mouse moves over the icon. The 2nd is the name of the file containing the icon, and the 3rd is the zero based index of the icon to use.

show	**SW**
	or
	(H, SW)
		dummy	Shows or hides the window. The value of the parameter can be: -
			
			`SW_HIDE 0`
			`SW_SHOWNORMAL 1`
			`SW_NORMAL 1`
			`SW_SHOWMINIMIZED 2`
			`SW_SHOWMAXIMIZED 3`
			`SW_MAXIMIZE 3`
			`SW_SHOWNOACTIVATE 4`
			`SW_SHOW 5`
			`SW_MINIMIZE 6`
			`SW_SHOWMINNOACTIVE 7`
			`SW_SHOWNA 8`
			`SW_RESTORE 9`
			`SW_SHOWDEFAULT 10`
			`SW_FORCEMINIMIZE 11`
			`SW_MAX 11`
			
			The second form allows the show state of another window to be addressed.

disableclose	**nil**	**dummy**	Eliminates the close button at the top right of a window, and from the system menu.
getwinver	**nil**	**tuple**	Returns the major and minor versions, build no, and the value returned by the GetVersion() API of the Microsoft Windows operating system.

			(getwinver()) 0 < 5 is true for WIN9x and 6.2 means Windows 8
wsize	**4-tuple**	**dummy**	wsize(x_{org}, y_{org}, x_{width}, y_{width}), Sets the window size to the parameters x_{width}, y_{width}, and moves the initial displaying position to x_{org}, y_{org} Parameters are expressed in twids, and are relative to the top left hand corner of the windows desktop.
getwsize	**nil** **0** **Non-zero**	**(l, t, r, b)**	Returns the screen coordinates (in twids) of the upper-left and lower-right corners of the currently selected output window. (uses GetWindowRect in WIN32 API). Bottom corner is outside window. Includes borders, title bar etc. If 0 is passed as an argument the outer parent's window size is returned. If a non-zero value is passed it is the handle of a window whose size is retrieved.
clienttoscreen	**(x, y)**	**(X, Y)**	Returns the absolute screen position in twids of the client coordinates in twids passed as arguments.
startup	**()**	**(X, Y, W, H)**	Window position and size passed to the GTL execution in the STARTUPINFO structure (in twids).
sizechange	**ref**	**dummy**	A non-zero argument causes subsequent changes to the window size to be communicated via the wmcommand() operator. Every time the window size is changed, three ref values are returned via the wmcommand buffer: the argument, and the new width & height of the window in twids.
activate	**ref**	**dummy**	A non-zero argument causes the value to be sent via wmcommand() every time a WM_NCACTIVATE message is processed by the Windows message loop.
getclientsize	**nil**	**(w, h)**	Returns the size in twids of the currently selected output window's client area (uses GetClientRect in WIN32 API)
icon	**string**	**string**	Changes the icon for the windows class (the main GTL window and any child windows). The argument is a string giving path and filename for a **.ico** file to use. The result is the null string upon success or an error string upon failure.
child	**(t, f, x, y, x_w, y_w)** **or** **(t, f, x, y, x_w, y_w, f2)**	**child**	Creates a child window. Argument **t** is a string to used in the title bar, **f** is a ref value of bit flags affecting the new window, and **(x, y)** are the twid coordinates of the top left hand corner of the child window relative to the client area of the parent (main GTL execution) window. **(x_w, y_w)** are the size of the window. **f2** is an optional parameter that allows the user to specify extended window creation flags. The **child** value returned by the child operator may be passed to the **select** operator to cause subsequent output to appear in the child window. Use (**select** 0) to reset the output to the parent window. The child window and associated memory usage is garbage collected once the value is no longer referred to from the GTL variable, and the window has been closed by the **close** operator. If a child window has been closed via the menu, or the X in the top right hand corner, then calling wmcommand will result in a value of –1.
setwindowflag	**ref**	**dummy**	Sets one or more of the bit flags in the Flags member of the oWindow structure of the main GTL execution window. The operator uses an XOR operation, so successive calls will toggle a particular flag value or values. The first 2 bits are reserved for specifying output layers. wF_Menu 4 Want menu wF_KillProcess 8 Kill whole process on window close wF_Seconds 16 Create 1 second interval timer wF_PageFormat 32 We are processing page formatting data wF_NoMargins 64 Deduct non-printable margins when printing wF_AutoScroll 128 Always want to see caret so

					scroll
			wF_Quietly	256	Suppress chimes
			wF_Hide	512	Hide window till output attempted
			wF_NoCaret	1024	Hide/Show the Caret
			wF_TrayIcon	2048	Show a icon in the system tray
			wF_Child	4096	This is not the parent window
			wF_CloseOnLoseFocus	8192	Window closes if it loses the focus
			wF_ChildFocus	16384	Indicates when a child's child has the focus
			wF_CloseFindOnFocus	32768	Close Find/Replace dialog when it loses the focus
			wF_ChildHasTimer	65536	Child window has a timer
			wF_DependOnAction	131072	Accept LSAActionExit message as an instruction to exit
			wF_DontCaptureFocus	262144	Don't take the focus
			wF_ToolWindow	524288	Tool window without top most setting
			wF_NoClose	1048576	Disable the X in the top right hand corner
			wF_IsClosed	2097152	I'm closed, no more output accepted
			wF_FinalExit	4194304	All windows processing - DestroyWindow is complete
			wF_NotifyAllMouseMsg	8388608	Send all mouse click messages regardless of capture, but only when not captured.
			wF_HScroll	16777216	Want Horizontal Scroll Bar
			wF_VScroll	33554432	Want Vertical Scroll Bar
			wF_Rubber	67108864	Rubberband capture mode on
			wF_AllKeyStrokes	134217728	Do not interpret keystrokes for scrolling, pass them to execution thread
			wF_NSSizing	268435456	Want a horizontal sizing line
			wF_WESizing	536870912	Want a vertical sizing line
			wF_EnableSizing	1073741824	Rectangles with wP_Capture want sizing cursors
			wF_Transparent	2147483648	The TransparentColour attribute contains valid information
			If toggling wF_Seconds off, then you need to call set_title to set the title of the window. If toggle wF_Seconds on, there will be a 1 second delay before the title is updated with the timer.		
setwindowflag2	**ref**	**dummy**	Similar to setwindowflag acting on a second 32-bit set of bitflags.		
			wF2_NoGraticule	1	Suppress graticule lines in sized rectangles
			wF2_InvalidateScroll	2	Full client area invalidate when scrolling.
			wF2_PrintLeftPages	4	Only print left pages - i.e. when x origin of page is 0, obsolete
			wF2_ToolWindow	8	Toolwindow, menu
			wF2_ActiveToMouseIn	16	Send Mouseclicks to mousein buffer for mouseactive objects, instead of wmcommand buffer
			wF2_TopMost	32	Topmost
			wF2_Link	64	Mark Text as Document designer link

			wF2_MinimiseNotify 128 Causes a -3 value to be sent via wmcommand() operator to notify that the window has been minimised, -5 for a restore, and -6 for a maximise. wF2_NoMouseScroll 256 Don't scroll when user uses the mouse wheel wF2_EMFOutput 512 Display primitives at 600 dpi wF2_NoPrinterDefault Don't attempt to set printer defaults. wF2_ExitIntercept For the outer parent window, causes -1 to be returned from wmcommand() instead of closing the window. wF2_ClearCode The mouseactive code is only for clear purposes. wF2_ParentControl mouseactive wmcommand codes are passed to the parent window. wF2_ChildStyle Create a child window with WS_CHILD style. Suitable for laying out child windows which track the positioning of the parent. Note a GTL child window is rather different from a Microsoft child window normally, but this flag brings them more into consistency. wF2_GlobalRubberBand Left button down presents rubber band rectangle at all times. On left button up wmcommand returns 9 values (-4, L,T,R,B,X0,Y0, X1,Y1) The LTRB define a rectangle and the X0,Y0,X1,Y1 provide direction for line drawing.
setwindowflag3	**ref**	**dummy**	wF3_LeaveFocus graphic elements are not to take focus when clicked. wF3_BMPOutput We are drawing in a memory DC bitmap wF3_VKHomeEnd Transmit Home as esc[H & End as esc[E wF3_FocusToParent Give the focus to the parent wF3_ToolWindow Tool Window without Topmost setting wF3_ExitIntercept Like wF2_ExitIntercept except applies to child windows as well. wF3_Arrows Pass arrow keys as 28, 29, 30, 31 ascii wF3_MouseDown Return mouseactive code on left mouse down
getwindowflag	**dummy**	**ref**	Returns the window flags for the currently selected window.
getwindowflag2	**dummy**	**ref**	Returns the window flags2 for the currently selected window.
sleep	**ref**	**dummy**	Suspends GTL execution thread for specified number of mS. For use with kbtraffic & wmtraffic, sleep 50 is good, to get good response without excessive CPU utilisation.
openbrowsewindow	**2-tuple** **or** **3-tuple**	**string**	Opens a standard Microsoft Windows file browse dialog box, and allows the user to select one file. The first argument is a tuple of string pairs. e.g. `("Comma Separated Value Files (*.csv)", "*.csv",`

	or **4-tuple**		`"Text Files (*.txt)", "*.txt")` The second argument is the path. The file name selected by the user is then returned. If the operator clicks the cancel button, a null string is returned e.g. let fname = openbrowsewindow(("Comma separated value files", "*.csv"), (envvar "USERPROFILE")."\Desktop") in The optional 3rd argument may be a default file name. The optional 4th argument allows extra flags to be passed to the Microsoft API such as OFN_ALLOWMULTISELECT, but that part of the API is very buggy.
browseforfolder	**(Title, CSIDL, Flags)**	**string**	Opens a Shell Dialog to allow the operator to select a folder. The selected folder is returned. The 2nd argument allows a root folder to be specified 0 -> Desktop. The third argument allows for flag settings in the SHBrowseForFolder API
getsavefilename	**3-tuple**	**string, ref**	Opens a standard "Save As" dialog. Generally similar to openbrowsewindow, the additional 3rd argument is the filename to prompt with. If the operator clicks Cancel, a zero length string is returned. An optional 4th argument permits the specification of a default extension. The return values are the file path string and the filter index selected by the operator.
screensize	**()**	**2-tuple**	Returns the working area of the primary monitor in twids.
monitorsize	**()**	**2-tuple**	Returns the full size of the primary monitor in twids.
virtualmonitor	**()**	**2-tuple**	Returns the combined size of multiple monitors in twids.
sendmessage	**(Msg, Wp, Lp)** or **(Hwnd, Msg, Wp, Lp)**	**ref**	Calls the Windows API SendMessage function with the specified parameters for the currently selected output and returns the result. *sendmessage* does not return until the message has been processed. The 2nd form permits a control handle to be passed as the 1st argument.
postmessage	**(Msg, Wp, Lp)** or **(Hwnd, Msg, Wp, Lp)**	**dummy**	Calls the Windows API PostMessage function with the specified parameters for the currently selected output window. *postmessage* queues the message and returns immediately, The 2nd form permits a window handle to be passed as the 1st argument.
findwindow	**(s1, s2)**	**ref**	Return a handle to a window by calling the API FindWindow function. E.g. `findwindow("LENNOX", "")` will return the handle of a L32 window.
handle	**()** **child**	**ref**	Returns the handle of the currently selected window or a child window. Useful for passing to a GTL process run separately as a tool window of some sort e.g. 3DControls, or Numeric Key Pad.
setfocus	**ref**	**dummy**	The argument is a window handle and the focus is given to that window.
focused	**()**	**ref**	Returns 1 if the currently selected window presently has the keyboard focus.
bringwindowtotop	**() or ref**	**dummy**	Brings a window to the top of the Z-order, without restore or focus changes.
setforeground	**()**	**dummy**	Brings the window to the top does a show restore and gives it the focus.
setforeground	**ref**	**dummy**	Brings the specified window to the top and gives it the focus (no show restore). Note **`setforeground LILACParent`** is the way to give the focus

			to LILAC.exe such that tool tips work. If ref arg is zero the current window is set foreground without a SW_RESTORE
getforegroundwindow	**()**	**ref**	Returns a ref value which is the handle of the window which has the focus.
enumwindows	**()**	**tuple**	Each member of the tuple contains data about a top level window: (H, T, P, C). H is the window handle, T is its title bar text, P is the Process Id and C is the Class Name.
enumchildwindows	**ref**	**tuple**	Arg is a window handle, result is $((h_0, t_0),(h_1, t_1), \ldots)$ where the h's are handles, and the t are window title of the child windows of the argument window.
createcontrol	**tuple**	**control**	Creates a child control. Arguments are (extended style, control class, window title, style, x, y, width, height, id) Id should be the **mouseactive** code for the control. The parent window's current font is set in the control window.
destroycontrol	**control**	**dummy**	Destroys a child control.
getrichtext	**ref**	**string**	Where the handle has been obtained from a Rich Text Edit **control** using the handle operator. The RTF data from the control will be returned with all the RTF formatting information. You can send RTF to a Rich Text Edit control with a simple WM_SETTEXT
setrichfont	**ref**	**dummy**	Communicates the current font settings to a rich edit control.
getrichfont	**ref**	**(f, p, w, c)**	Where f is the Type Face, p is Points, w is zero, and c is (R,G,B)
richposition	**(h, x, y)**	**p**	**h** is handle of a control, **x, y** are coordinates within the control and **p** is the corresponding character position.
getid	**control**	**ref**	Gets the id of child control.
getcontrol	**ref**	**control**	Gets a handle to a child control from an id. If a child control corresponding to the id can not be found a ref 0 is returned.
dragacceptfiles	**ref**	**dummy**	Turns on accepting files by drag and drop into the currently selected window. The argument will be returned via **wmcommand()** when one or more files are dropped on the window.
dragfiles	**( )**	**tuple**	May only be called after a **wmcommand()** returns the ref arg passed to **dragacceptfiles**. Returns a tuple of strings being the full paths to the file(s) dropped on the window.
dragpoint	**( )**	**(x, y)**	Coordinates in twids of drop position in window. To he called after a **wmcommand()** returning ref arg passed to **dragacceptfiles**
zoom	**num**	**dummy**	Default setting is zoom 1.0, values > 1.0 magnify the *vps* and values < 1.0 shrink the *vps* presentation.
printzoomed	**ref**	**dummy**	Apply the zoom factor in a printer device contact as well as in the *vps.*
windowrect	**ref**	**(l,t,r,b)**	Arg is a window handle - returns the window's rectangle in screen twids.
isiconic	**ref**	**0/1**	True if window is minimised,
flashwindow	**()**	**dummy**	Invokes FlashWindow API call

CD Burning Operators
CD burning in GTL is only supported under Windows XP (or later), or computers that have the Nero Burning ROM 5.5 software installed.

cd_burn	**t** or **(t, s)** or **(t, s, r, pb)**	**string**	Burns the cd. t is a tuple of absolute file paths which are to be burnt to the disc. s is the label of the cd. r is a "boolean" ref which indicates if the burn process should be simulated. pb is a ref progress bar variable. Returns "" on success and an error message string on failure. When cd_burn returns, it releases all resources associated with the cd burner.
cd_getdrives	**nil**	**tuple**	Returns a tuple of all CD Burner drives GTL can use. If nil is returned, it means no supported CD burner drives were found. For more information call cd_getlasterror.
cd_getlasterror	**nil**	**string**	Returns the last CD error.
shellcd_burn	**nil**	**string**	Supported only Windows XP or later. Runs the shell's CD Burning Wizard. Any files to be burnt to the cd must be copied to the staging area. This can be done by either using shellcd_addfile or by retreiving the staging path (shellcd_getstagepath), and copying the files there. Returns "" on success and an error message string on failure. When shellcd_burn returns, it releases all resources associated with the cd burner.
shellcd_getstagepath	**nil**	**string**	Supported only Windows XP or later. Returns the staging path on success and "" on failure. For more information about any errors that occur use shellcd_getlasterror.
shellcd_addfile	**string**	**string**	Supported only Windows XP or later. Copies the file the the staging area. Returns "" on success and an error message string on failure.
shellcd_getdrives	**nil**	**tuple**	Supported only Windows XP or later. Returns a tuple of all CD Burner drives GTL can use. If nil is returned, it means no supported CD burner drives were found. For more information call cd_getlasterror. Because the shell only supports one CD Burner, the tuple returned with only ever be of order 0 or 1.
shellcd_getlasterror	**nil**	**string**	Returns the last CD error.

DVD Burning Operators
DVD burning in GTL is only supported on computers that have the Nero Burning ROM 6.0 software installed.

dvd_burn	**t** or **(t, s)** or **(t, s, r, pb)**	**string**	Burns the dvd. t is a tuple of absolute file paths which are to be burnt to the disc. s is the label of the dvd. r is a "boolean" ref which indicates if the burn process should be simulated. pb is a ref progress bar variable. Returns "" on success and an error message string on failure. When dvd_burn returns, it releases all resources associated with the dvd burner.
dvd_getdrives	**nil**	**tuple**	Returns a tuple of all DVD Burner drives GTL can use. If nil is returned, it means no supported DVD burner drives were found. For more information call cd_getlasterror.
dvd_getlasterror	**nil**	**string**	Returns the last DVD error.

Registry Operators

lsareg	**(p, k)**	**string** or **tuple**	The parameters are the product and key. Returns a "" if the key or product does not exist else it returns the value of the key. The section of the registry inspected is **HKEY_CURRENT_USER\Software\Lennox Computer** A tuple is returned when lsareg encounters an ETR stored as binary value in the registry.
lsaset	**(p, k, v)**	**dummy**	The parameters are the product, the key and the value of the key. It places these values in the registry, creating any needed folders. As necessary the product is placed under **KEY_CURRENT_USER\Software\Lennox Computer** in the registry **p** & **k** are strings and **v** may be a string or a tuple. If **v** is a tuple it is converted to an ETR and stored as a binary value in the registry.
hlmreg	**(p, k)**	**string** or **tuple**	The parameters are the product and key. Returns a "" if the key or product does not exist else it returns the value of the key. The section of the registry inspected is **HKEY_LOCAL_MACHINE\Software\Lennox Computer** A tuple is returned when lsareg encounters an ETR stored as binary value in the registry.
hlmset	**(p, k, v)**	**dummy**	The parameters are the product, the key and the value of the key. It places these values in the registry, creating any needed folders. As necessary the product is placed under **HKEY_LOCAL_MACHINE \Software\Lennox Computer** in the registry **p** & **k** are strings and **v** may be a string or a tuple. If **v** is a tuple it is converted to an ETR and stored as a binary value in the registry.
regdeletekey	**(h, s)**	**string**	Deletes a key from the Windows Registry
regdeletevalue	**(h, s)**	**string**	Deletes a value from the Windows Registry
regcreatekey	**(h, s)**	**ref**	Creates the specified registry key. If the key already exists, the function opens it. The first parameter is a ref which is a handle to an open key. This handle is returned by the regcreatekey or regopenkey, or it can be one of the following predefined keys: HKEY_CLASSES_ROOT HKEY_CURRENT_CONFIG HKEY_CURRENT_USER HKEY_LOCAL_MACHINE HKEY_USERS The second parameter is a string containing the name of the subkey to create. Returns zero if the registry key is not created or opened, else returns a handle to the key as a ref value. e.g. **let Software = regcreatekey(HKEY_CURRENT_USER, "Software") in**
regopenkey	**(h, s, w)**	**ref**	Opens the specified registry key. The first parameter is a ref which is a handle to an open key. This handle is returned by the regcreatekey or

			regopenkey, or it can be one of the following predefined keys: HKEY_CLASSES_ROOT HKEY_CURRENT_CONFIG HKEY_CURRENT_USER HKEY_LOCAL_MACHINE HKEY_USERS The second parameter is a string containing the name of the subkey to open. The third parameter is a ref 1 for write access, 0 for read only access. Returns zero if the registry key is not found, else returns a handle to the key as a ref value. e.g. `let Software = regopenkey(HKEY_CURRENT_USER, "Software", 0) in`
regclosekey	**h**	**dummy**	Close a registry key

regenumvalue	**(h, i)**	**(Name, Data)**	e.g.
			`let ND = regenumvalue(h, i) in`
			`unless null ND do`
			`{ let Name, Data = ND in`
			The 2nd argument should be incremented from 0 to enumerate all the values of a key. The result is a 2-string tuple, or null if there are no more values.

regenumkey	**(h, i)**	**tuple**	i should be 0 for the 1st key
regsetvalueex	**(h, k, v)**	**string**	Returns null string on success.
regsetvalue	**(h, k, v)**	**string**	Returns null string on success.

Progress Bar Operators

progressbar	**tuple**	**dummy**	Takes a tuple of format (height, width, lv), where height is the desired height in twids, width is the desired width in twids, and lv is a variable which will be used to increment the progress bar. The value of the variable is monitored in a Windows timer message, so it just needs to change arithmetically in the range 0 to 99 to animate the progress bar. Use *close* to dispense with a progress bar, which will also remove it from the display.

Zip Operators

zipclose	**zip**	**Dummy**	Closes the open zip file
zipadd	**(zip, f, pb)**	**ref**	Adds a file specified by string **f**.. Returns the number of files successfully added to the archive (always 1). **pb** is a ref progress bar variable. The path passed as **f** is recording in the zip, UNC paths with \\ are recorded as folders in the zip.
zipextractall	**(zip, p)**	**ref**	Extracts all files in the zip to the path specified by the parameter **p**. Returns 1 for success.
zipopen	**(f, flag)**	**zip**	Opens an existing zip file with the name passed in by the first parameter. If **flag** =0 open an existing file, fails if file doesn't exist and **flag** = 2 -> create a new file, fails if the file does exist. On failure the return value is a string error message.

Miscellaneous Operators

run	**(c, d, t)** **(c, d, t, S)**	**string**	Parameters are the command line, the directory to run the process in, and the length of time you want to wait in mS before continuing. If the length of time to wait is negative, then it waits until the process exits The target process may be an **.exe** file in which case *run* uses the CreateProcess API to invoke it, or it may be a document in which case *run* uses ShellExecute.

	(c, d, t, S, X, Y, W, H)		The optional 4th argument is an SW_SHOW value. X, Y, W, H are optional position and size settings in twids for the target process. A GTL program uses the **startup** operator to retrieve these. *run* returns the null string to indicate success and an error string to indicate failure.
creationflags	**ref**	**dummy**	e.g. creationflags IDLE_PRIORITY_CLASS Use this operator prior to a call to **run** above to set the Creation Flags for the Windows CreateProcess API.
lastpid	**()**	**ref**	Returns the PID of the process created by the last *run* or *spawn* operator.
runwait	**(s, W)**	**dummy**	Runs a subsidiary process – **s** is the executable concatenated with the rest of the command line. GTL waits for the subsidiary process to exit before continuing. **W** is the SW_SHOW parameter desired in range 0 to 11, use -1 to use the default show. SW_HIDE = 0
runtimeout	**(s, W, t)**	**dummy**	Same as runwait, but with a timeout t in mS
runwxy	**(s, t, x, y)**	**dummy**	Run a process at a position with a timeout.
gtl	**(s, W, t)**	**tuple**	**s** is the gtl executable concatenated with rest of the command line. **W** is the SW_SHOW parameter, and **t** is a tuple of arguments to be passed the subsidiary process via a temporary file. The return value is a tuple obtained from the subsidiary GTL process calling **etrreturn** via a temporary file created in the Local App Data folder e.g. `let t = gtl ("Hello.gtl", SW_SHOWNORMAL,()) in`
suspend	**0/1**	**dummy**	Suspends message processing for the current selected window. i.e. messages are discarded. Useful when a foreground window has the focus invoked via the **gtl** operator, and the background window shouldn't respond until the gtl invoked process returns.
etrarg	**PID**	**tuple**	Used by a subsidiary process to retrieve the argument tuple passed as the 3rd argument to **gtl** above**. PID** is the Process Id of the calling process which the **gtl** operator appends as the final white separated argument to the command line used to invoke the subsidiary process.
etrreturn	**tuple**		The tuple is converted to an ETR written to a temporary file in the Local App Data folder for retrieval by the **gtl** operator in the calling GTL process. The subsidiary process then terminates.
runresult	**(c, d)**	**ref** **string**	Parameters are the command line, the directory to run the process in. Returns the exit code. If the process fails to run a string rv is returned.
spawn	**(s, W, t)**	**dummy**	Same behaviour as the **gtl** operator above except no data returned and calling process runs independently. Operator returns immediately. *lastpid* operator returns PID of spawned process.
runaswait	**(tk, string)**	**dummy**	Parameters are the token obtained from logon, the executable concatenated with the rest of the command line. GTL waits for the subsidiary process to exit before continuing.
runelevated	**(e, p, d)**	**string**	Uses shellexecute to invoke executable **e** with elevated privileges (Administrator) – Vista & later will prompt for Administrator password. **p** is parameters, **d** is directory. For example: `runelevated("gtl.exe", "Associations.gtl", getcurdir())`

shellexecute	**(v, f, p, d, s)**	**string**	v (verb) from ("open", "print", "printto", etc)
			f windows document file e.g. Letter.doc
			p parameters to be passed – e.g. name of printer
			d directory to run in
			s ref value for show command. e.g.
			`SW_HIDE 0`
			`SW_SHOWNORMAL 1`
			`SW_SHOWMINIMIZED 2`
			`SW_SHOWMAXIMIZED 3`
			`SW_SHOWNOACTIVATE 4`

			SW_SHOW 5 SW_MINIMIZE 6 SW_SHOWMINNOACTIVE 7 SW_SHOWNA 8 SW_RESTORE 9 SW_SHOWDEFAULT 10 SW_FORCEMINIMIZE 11 Returns the null string to indicate success, and an error string to indicate failure.
shortcut	**tuple**	**dummy**	This operator creates a windows shortcut. This operator can take 7 or 9 parameters. The first argument is a **string** that specifies the module to execute. The second argument is a **string** that specifies the shortcut filename. The third argument is a **string** that specifies that contains a description of the shortcut. The fourth argument is a **string** that specifies the initial working path. The fifth argument is a **string** that specifies the parameters to pass to the module to execute. The sixth argument is a **ref** that specifies the initial window position. This can be either 1 (normal), 3 (maximized), or 7 (minimized). The seventh argument is a **ref** that specifies if the shortcut should be created with the default icon, or with a custom icon. Specify FALSE to use the default icon. If you specify TRUE and the number of arguments is 7, then the icon used is the zeroth icon in the file specified by the first argument. If however the number of arguments is 9, then the eighth and ninth arguments are used to specify the icon. The eighth argument is a **string** that specifies the path to the icon file to be used. This can be an EXE, DLL, or ICO file. The ninth argument is a **ref** that specifies the zero based offset of the icon in the icon file. If the icon offset is less than 0 or greater than the number of icons in the icon file, then a default windows application icon is used. These parameters are ignored if use icon is 0 `shortcut ("c:\lilac3\l32.exe", envvar("USERPROFILE"). "\Start Menu\Programs\Lilac 3 Client.lnk", "Lilac 3 Client", "c:\lilac3", "test", 3, 1);` This creates a short cut in the start menu.
createzone	**string**	**dummy**	e.g. createzone "\\lilac-server" Adds an entry to the intranet zone for the local computer.
faxdocument	**tuple**	**tuple**	The 1st argument is the recipient's name. The 2nd argument is the recipient's number. The 3rd argument is the sender's name. The 4th argument is the sender's number. The 5th argument is the layout of the document. TRUE for landscape, FALSE for portrait. It is assumed that a standard Microsoft fax printer driver is installed on the computer. This operator only works under Windows 2000 and later. Returns a tuple in format (ref, string, ref). This first member is FALSE if winfax.dll is not loaded. In this case the program should terminate after performing any clean operations. The second member returns any error string. The third member returns the job id.
faxprinter	**string**	**dummy**	Establishes a faxprinter other than the default local faxprinter.

mapisend	**(s, b, t, A, Batch)**	**ref**	Where s is the subject, b is the body text and t is a tuple of recipients of the form ((“Doug Lennox”, SMTP:doug@lennox.com.au, 1), (“Ailsa Lennox”, SMTP:ailsa@lennox.com.au, 2), . . .) and A is a tuple of attachment full paths. If Batch is true, no dialog box is displayed. Return value is 0 for success, or a MAPI error code.
getfaxdocuments	**nil**	**tuple**	The return value is a tuple of tuples in format (Job Id, Fax Type, Document Name, Status, Destination, Destination Fax Number, Sender, Sender Fax Number, Number of Pages, Schedule).
setfaxdocuments			obsolete
enumprinters	**ref**	**tuple**	The argument specifies which printer info data structure to use. Possible values are 2, 4, 5. Returns a tuple of 4-tuples of the form (printer name, server, attributes, port) for all printers local or remote, visible on the PC. It will return a string in the event of an error.
printertray	**ref**	**dummy**	Specify a printer tray selection e.g. 257 may be Tray1
devicecapabilities	**(string, string, ref)**	**tuple**	The 1st argument is the printer name, the 2nd argument is the port (e.g. “LPT1:”), the 3rd argument specifies which capabilities to query. Returns a tuple. Exactly what is returned differs depending on the capability specified. Please see the Microsoft Platform Documentation for more information. It will return a string in the event of an error.
exitin	**ref**	**dummy**	Specifies the number of seconds to wait before exiting the GTL process. Pass exitin –1 as the parameter to specify infinity.
drill	**tuple**	**dummy**	
drilldoc	**tuple**	**dummy**	Drill down to a LILAC WYSIWYG document. The arguments are: (Process, Command, Key1, Key2, User, Document, PDFflag)
finished	**0/1**	**dummy**	*finished 1;* sets the GTL window such that the close or exit windows command takes immediate effect without a caution dialog box when the GTL thread is still running.
createservice	**(n, d, b, D)**	**string**	Provides the ability to create entries in the Windows Service database. Require Administrator privileges. **n** is the name of the service, **d** is the display name, **b** is the binary path, **D** is the details annotation. If the service pre-exists, an error is returned, call createservice with a null string as the **b** argument to delete a pre-existing service.

Direct X Operators

GTL uses untransformed and unlit vertices. By using untransformed and unlit vertices, GTL requests that Microsoft® Direct3D® perform all transformation and lighting operations using its internal algorithms.

You are required to specify vertices in untransformed model coordinates. The system then applies world, view, and projection transformations to the model coordinates to position them in your scene and determine their final locations on the screen.

d3_window	**tuple**	**directx**	Creates a direct x child window. The format of the tuple passed as the argument is (x_{org}, y_{org}, x_{width}, y_{height}). The window by default is not displayed. To show the window, output the **directx** rvalue.
d3_clear	**tuple**	**dummy**	Clears the presentation space in the direct x child window. The format of the tuple passed as the argument is (**directx, ref**). The 2nd argument is a *d3_makecolour* value.
d3_destroy	**directx**	**dummy**	Destroys the direct x window, and deal locates the appropriate direct x data structures.
d3_present	**directx**	**zip**	Displays everything that was in the Direct X Backbuffer to the screen.
d3_makecolour	**tuple**	**ref**	Returns a 32bit unsigned integer that represents an RGBA colour. The format of the tuple passed as the argument is (r, g, b, a)

The following operators all take the same arguments. The format of the parameter passed is (**directx**, **ref**, **tuple of points**).

The **directx** parameter specifies which Direct X window to use.
The **ref** parameter specifies the number of objects to be drawn.
The **tuple of points** is a tuple of (x, y, z, colour) formatted tuples.

d3_point	**tuple**	**directx prim**	Returns a Direct X Primitive. This can be outputted to add it to the Direct X Backbuffer.
d3_line	**tuple**	**directx prim**	Returns a Direct X Primitive. This can be outputted to add it to the Direct X Backbuffer.
d3_triangle	**(d3, n, t)**	**directx prim**	Returns a Direct X Primitive. This can be outputted to add it to the Direct X Backbuffer.
d3_trianglestrip	**tuple**	**directx prim**	Returns a Direct X Primitive. This can be outputted to add it to the Direct X Backbuffer. You can only specify one triangle strip per call.
d3_trianglefan	**tuple**	**directx prim**	Returns a Direct X Primitive. This can be outputted to add it to the Direct X Backbuffer. You can only specify one triangle fan per call.
d3_addindex	**tuple**	**dummy**	Adds an index to a Direct X Primitive.
d3_settexture	**(dp, string)**	**dummy**	Adds a texture to a Direct X Primitive. Arguments are **Directx Primitive** to apply the texture. And the **string** is a path to a BMP file.

The following operators defines lighting properties.

Diffuse
Diffuse color emitted by the light. Ceated by using the d3_makecolour operator.

Specular
Specular color emitted by the light. Ceated by using the d3_makecolour operator.

Ambient
Ambient color emitted by the light. Ceated by using the d3_makecolour operator.

Position
Position of the light in world space. This has no meaning for directional lights and is ignored in that case. Consists of X, Y, Z components.

Direction
Direction that the light is pointing in world space. This member only has meaning only for directional and spotlights. This vector need not be normalized, but it should have a nonzero length. Consists of X, Y, Z components.

Range
Distance beyond which the light has no effect. The maximum allowable value for this member is the square root of FLT_MAX. This member does not affect directional lights.

Falloff
Decrease in illumination between a spotlight's inner cone (the angle specified by **Theta**) and the outer edge of

the outer cone (the angle specified by **Phi**). The effect of falloff on the lighting is subtle. Furthermore, a small performance penalty is incurred by shaping the falloff curve. For these reasons, most developers set this value to 1.0.

Attenuation0, **Attenuation1**, and **Attenuation2**

Values specifying how the light intensity changes over distance. Attenuation values are ignored for directional lights. These members represent attenuation constants. Valid values for these members range from 0.0 to infinity. For non-directional lights, all three attenuation values should not be set to 0.0 at the same time.

Theta

Angle, in radians, of a spotlight's inner cone—that is, the fully illuminated spotlight cone. This value must be in the range from 0 through the value specified by **Phi**.

Phi

Angle, in radians, defining the outer edge of the spotlight's outer cone. Points outside this cone are not lit by the spotlight. This value must be between 0 and *pi*.

d3_pointlight	**(d3, index, diffuse, ambient, specular, position, Attenuation, range)**	**dummy**	Creates and enables a point light in a Direct X window.
			
			`d3_pointlight(d3, 0,`
			`d3_makecolour(255, 255, 255, 125),`
			`d3_makecolour(0, 0, 0, 255),`
			`d3_makecolour(0, 0, 0, 255),`
			`0.0, 0.0, -10.0, 1.0, 0.0, 0.0, 100.0);`

d3_spotlight	**(d3, index, diffuse, ambient, specular, position, attenuation, range, direction, Falloff, Theta, Phi)**	**dummy**	Creates and enables a spot light in a Direct X window.
			
			`d3_spotlight(x, 0,`
			`d3_makecolour(255, 255, 255, 125),`
			`d3_makecolour(0, 0, 0, 255),`
			`d3_makecolour(0, 0, 0, 255),`
			`0.0, 0.0, -10.0,`
			`1.0, 0.0, 0.0, 100.0,`
			`-0.5, -1.0, 1.0,`
			`1.0, 1, 2);`

d3_directionallight	**(d3, index, diffuse, ambient, specular, direction)**	**dummy**	Creates and enables a directional light in a Direct X window.
			
			`d3_directionallight(x, 0,`
			`d3_makecolour(255, 255, 255, 125),`
			`d3_makecolour(0, 0, 0, 255),`
			`d3_makecolour(0, 0, 0, 255),`
			`-0.5,-1.0, 1.0);`

d3_ambientlight	**(d3, ref)**	**dummy**	Turns on ambient lighting for a Direct X window. The format of the tuple is (d3, colour) where colour is created using d3_ makecolour.
d3_enablelight	**(d3, ref, ref)**	**dummy**	Enables or disables the light at the given index. The format of the tuple argument is (d3, index, enabled). If enabled is 0 then the light is switched off, else it is switched on.
d3_setmaterial	**(dp, ref, ref, ref, ref, ref/num)**	**dummy**	Sets the material for a given direct x primitive. The format of the tuple is (dp, diffuse, ambient, specular, emissive, power) where diffuse, ambient, specular and emissive are created using the d3_makecolour operator.
d3_deleteallprims	**d3**	**dummy**	Deletes all the primitives in the given direct x window.
d3_deleteprim	**dp**	**dummy**	Deletes a single direct x primtive.
d3_translate	**tuple**	**dummy**	Translates a Direct X Primitive in the Direct X Window. Format of argument is (dp, x, y, z)
d3_rotatex	**tuple**	**dummy**	Rotates a Direct X Primitive on its X axis. Format of argument is (dp, angle in radians)
d3_rotatey	**tuple**	**dummy**	Rotates a Direct X Primitive on its Y axis. Format of argument is (dp, angle in radians)
d3_rotatez	**tuple**	**dummy**	Rotates a Direct X Primitive on its Z axis. Format of argument is (dp, angle in radians)

d3_scale	**tuple**	**dummy**	Scales a Direct X Primitive in the Direct X Window. Format of argument is (dp, x, y, z)

Conditional Expression

Syntax

b -> e_1 | e_2

Description

The expression *b* is evaluated before either expression e_1 or expression e_2 is evaluated. The evaluation of *b* must return a *ref* value. If the value of *b* is not equal to 0, then the e_1 is evaluated and the value returned. If the value of *b* equals 0, then the e_2 is evaluated and its value returned.

Note that either e_1 or e_2 is evaluated, but never both.

Example 1

```
\Illustration of basic conditional operator use
let x = 10 in                 \declare the variable x and initialise to 10
let y = x > 10 -> 1 | -1 in   \declare the variable y. If x is greater than 10
                              \initialise the
                              \value of y to 1 (one)
                              \else initialise the value of y to -1
                              \(negative one)
y > 0 -> y | -y               \if y is greater than 0 display the value of y,
                              \else display the
                              \negated value of y
```

Output to Example 1

1

Example 2

```
\Illustration of a nested conditional operator
let diff = 4 in
let x = diff = 1 -> 100 |
        diff = 2 -> 900 |
        diff = 3 -> 1600 |
        diff = 4 -> 2300 |
        diff = 5 -> 3000 |
        diff = 6 -> 3700 | 4400 in
x
```

Output to Example 2

2300

Notes

Conditional operators maybe nested, as illustrated by example 2. Parentheses and indentation should be used to resolve ambiguity and improve legibility for the programmer.

Of course the GTL conditional expression is derives its syntax & semantics historically from the COND operator in LISP 1.5 and combined with the applicative lambda calculus evaluation facility of GTL provides a very powerful recursive function evaluation engine for the manipulation of symbolic data and artificial intelligence applications.

let Declaration

Syntax

let $n = e_0$ **in** e_1	\e_0 *may evaluate to a scalar* or a tuple*
let $n_1, n_2, .., n_k = e_0$ **in** e_1	\e_0 *must evaluate to a tuple of order k*
let $n\ p_1 = e_0$ **in** e_1	\ p_1 *may bind to scalar or a tuple when n is applied*
let $n(p_1, p_2..,p_k) = e_0$ **in** e_1	\ $p_1, p_2..,p_k$ *must bind to tuple of order k when n is applied*
let rec $n\ p_1 = e_0$ **in** e_1	\ p_1 *may bind to scalar or a tuple when n is applied*
let rec $n(p_1,p_2..,p_k) = e_0$ **in** e_1	\ $p_1, p_2..,p_k$ *must bind to tuple of order k when n is applied*

Description

The **let** statement provides for the definition of variables, and functions.

Immediately after the **let** key word there can appear a single variable name, a list of variable names, or a function name with formal parameter list.

Where *n* is a single variable name, e_0 may evaluate to any value. Where n is a list of variable names of the form n_1, n_2, …, $n_k = e_1$ then e_1 must evaluate to a tuple of size k, where n_1, n_2, …, n_k are the individual variable names.

If *n* is to a function name, then a single formal parameter may bind to a scalar or a tuple of actual parameters which provides for a form of polymorphic parameterisation. Where multiple formal parameters are specified then the number of actual parameters (order of the tuple) must match the number of formal parameters.

If the **rec** operator is used then this allows *n* to recursively call itself within e_0. If **rec** is not specified any reference to name *n* within e_0 are bound outside this **let** declaration.

The scope of the variable or function *n* is confined to the expression e_1. Expression e_1 can be as simple as single statement or it can be as complex as a thousand-line program.

Example 1

```
                        \Simple use of let to define variables.
let x, y = 1, 2 in      \Where variables are simultaneously
"x = "; x;              \defined like this, the expressions
" y = "; y              \to the right of the = sign are all
                        \evaluated before the variables are defined.
```

Output to Example 1

x = 1 y = 2

Example 2

```
                  \Simple example of using the let operator to
let f1 x =        \define a function which requires a parameter x
{     x * 3
} in
f1 3
```

Output to Example 2

9

Example 3

```
\self referential function definition

let rec Factorial n = (n = 0) -> 1 | n * Factorial(n-1) in
Factorial 6
```

Output of Example 3

720

(* In GTL, a “scalar” is any value which is not a tuple.)

Where Expression

Syntax

e_1 **where** $n = e_0$

e_1 **where** $n_1, n_2, .., n_k = e_0$

e_1 **where** $n\ \ p_1 = e_0$

e_1 **where** $n(p_1, p_2, .. p_k) = e_0$

e_1 **where rec** $n\ \ p_1 = e_0$

e_1 **where rec** $n(p_1, p_2, .. p_k) = e_0$

Description

The **where** statement provides for post definition of variables, and functions.

Immediately after the **where** a single variable name, a list of variable names, or a function name with formal parameter list may appear.

Where n is a single variable name, e_0 may evaluate to any value. Where n is a list of variable names of the form n_1, n_2, …, $n_k = e_1$ then e_1 must evaluate to a tuple of size k, where
n_1, n_2, …, n_k are the individual variable names.

If n is to a function name, then the syntax must be $n(p_1, p_2, \ldots, p_k) = e_0$ where
p_1, p_2, … p_k are the parameter names, n is the function name, and e_0 is an expression. If the **rec** operator is used then this allows n to recursively call itself within e_0.

The scope of the variable or function n is confined to the expression e_1. Expression e_1 can be as simple as single statement or it can be as complex as a thousand-line program.

Example 1

\Simple where statement

```
x                               \use x
where x = 1                     \now define x
```

Output to Example 1

1

Example 2

```
\Simple where statement
let y = 3 in                    \define y and initialise its value to 3
f1(y)                           \call procedure f1 with parameter y
where f1(x) =                   \define procedure f1 with a single parameter
{     x*x
}
```

Output to Example 2

9

Notes

Practical experience suggests programmers are generally more comfortable with **let** declarations and rarely deploy the **where** form.

The source management of large programs is easier when the programmer is assured that identifiers are declared before they are used i.e. earlier in the source file.

Lambda Expression

Syntax

fn *V1* **.** *E*

Description

A lambda expression denotes a function. The *V1* component is called the **bound variable**-part. The variables named in *V1* are called the **bound variables** of the lambda expression. Bound variables are those, which are to be substituted for in the expression *E*. when the lambda expression is applied to actual arguments.

Any variables referred to in *E* which are not defined in *V1* are called **free variables.**

The handling of the evaluation of **free variables** by the language interpreter is the most important aspect of the entire design of the GTL programming language.

The concepts involved are difficult for programmers who lack a strong theoretical background in the Theory of Computation.

The concept revolves around a “function object” - for example in the “C” language when a function name is passed as an argument to another function without actually calling the function. In “C”, any free variables in such a function will bind only to outer level global variables so there is no conceptual difficulty. In GTL, any free variables in a lambda expression must bind to the environment in which the lambda value is created. This means that the “function object” created by the evaluation of a lambda expression (note: *evaluation* does not mean *application* to actual parameter values) must carry a combination of the original lambda expression and an *environment*, which provides the values for any **free variables** in the lambda expression.

Example 1

```
(fn b . (fn a. a + b) 2) 37
```

Output to Example 1

39

Example 2

```
let b = 37 in
(fn b . (fn a. a + b) 2 + (fn g . ( fn (x, y) . (x + 1) * y) (g, b)) b - (fn c.
b * c + 2) b) b
```

Output to Example 2

74

Notes

In example 2, note that b in the lambda expression is bound by that example, and the fact that it already had a value is irrelevant.

Within definitions

Syntax

let d_1 = e_0 **within** d_2

Description

The declaration $d_1 = e_0$ is “private” to the declaration d_2 .

Most importantly variables declared in d_1 persist as long as a (lambda expression) declared in d_2, but are private to that lambda expression, and make no “holes” in the scope of globally defined variables using the same identifier.

This concept is the true lambda calculus implementation of “object oriented” programming. Variables bound “privately” to a lambda expression in this fashion provide a logically consistent and complete version of the rubbery “object oriented” concept, and the power and efficiency of the GTL interpreter and memory allocation engine make the concept work in the real world with lightning fast execution.

Example 1

\The 1st declaration of **n** is global, the 2nd is “private” to the lambda expression `Next`.

```
{     let n = 10 in
     {    let n = 2 within Next() = valof
          {     n += 1; res n
          } in
          for i = 0 to 9 in
          {     "Next() = "; Next(); ", n = "; n; NL
          }
     }
}
```

Output to Example 1

```
Next() = 3, n = 10
Next() = 4, n = 10
Next() = 5, n = 10
Next() = 6, n = 10
Next() = 7, n = 10
Next() = 8, n = 10
Next() = 9, n = 10
Next() = 10, n = 10
Next() = 11, n = 10
Next() = 12, n = 10
```

The **and** keyword is available to allow **within** variables to be privately associated with a group of lambda expressions declared simultaneously.

Valof and Res Commands

Syntax

valof *C*

res *E*

Description

The effect of obeying an expression such as **res** *E* is that the evaluation of the smallest enclosing **valof** is terminated, and the value of *E* is used as the value of the entire **valof**. Nesting of **valof** blocks inside one another is permitted. *C* is a sequence of commands.

Example 1

```
                          \a simple valof and res example.
let Abs x = valof         \define a function that returns absolute value of x
{  if x > 0 do res x;     \if x is greater than 0 return x
   res -x                 \else return negated x
} in
let i = -2 in
Abs I                     (note: lower case abs is a built in GTL operator)
```

Output to Example 1

2

Example 2

```
\a complex valof and res expression
let StateCode(x) = valof
{     let id =    x = "QLD" -> 0 | x = "NSW" -> 1 |
                  x = "VIC" -> 2 | x = "NT" -> 3  |
                  x = "SA" -> 4  | x = "TAS" -> 5 |
                  x = "WA" -> 6  |  x = "ACT" -> 7 | i
                  where i = valof
                  {     if (x = "OVERSEAS") do res 8 else res 9
                  }
      in
      res id
} in
let S1, S2 = "QLD", "UNKNOWN" in
StateCode(S1);
StateCode(S2)
```

Output to Example 2

09

Notes

The commands **valof** and **res** are intended to be used in conjunction with each another. If they are not however, and **res** is called without a corresponding **valof**, a *res* value is returned. If **valof** is called without a corresponding **res** command, then **valof** returns a *dummy* value.

reslv *E*

An alternative form of the res operator which will return the lvalue of the expression, so that it may be modified by the caller.

If Command

Syntax

if *B* **do** C_1
if *B* **do** C_1 **else** C_2

Description

The expression *B* is evaluated before either, command C_1 or command C_2 is executed. The evaluation of *B* must return a *ref* value. If the value of *B* is not equal to 0, then the result of the evaluation of C_1 is returned. If the value of *B* equals 0, then the result of the evaluation of C_2 is returned.

Example 1

```
let x = 1 in                    \declare a variable x and initialise it to 1
if x do x + 1                   \if x does not equal 0 then add 1 to x
else x - 1                      \else subtract 1 from x
```

Output to Example 1

2

Example 2

```
\Use an if statement to find the absolute value of x

let x = -1 in                   \declare a variable x and initialise it to -1
if (x < 0) do x := -x;          \if x is less than 0 then negate x
x
```

Output to Example 2

1

Unless Command

Syntax

unless *B* **do** C_1

Description

The expression *B* is evaluated before command C_1 is executed. The evaluation of *B* must return a *ref* value. If the value of *B* is equal to 0, then the result of the evaluation of C_1 is returned.

Example 1

```
let s = "ab" in
unless length s <> 2 do s
```

Output to Example 1

ab

While Statement

Syntax

while *b* **do** C_1

Description

The expression *b* is evaluated before the command C_1 is performed. The evaluation of *b* must return a *ref* value. While the value of *b* is not equal to 0, C_1 is repeatedly evaluated. The value of a while statement is always **dummy,** ie the same value returned by an assignment statement. The value of C_1 is output to the default output context, every-time it is evaluated.

Example 1

```
\demonstration of a simple while statement
let i = 10 in                 \declare a variable i and initialise it to be 10
while (i > 0) do              \while i is greater than 0
{     i; " ";                 \print out the value of i
      i := i - 1              \decrement the value of i
}
```

Output to Example 1

10 9 8 7 6 5 4 3 2 1

Example 2

```
\demonstration of a simple while statement
while (i > 0) do              \while i is greater than 0
{     if (i - x) do           \if i - x doesn't equal 0
      {     i := (i / x) + 1  \calculate the value of i
      };
      x := i - x;             \update the value of x
      "(";x;",";i;")"         \print out the value of x and i
}
where i, x = 10, 2
```

Output to Example 2

(4,6)(-2,2)(2,0)

Assignment Statement

Syntax

E_1 := E_2

Description

E_1 is evaluated to yield an L-value and E_2 is evaluated to yield an R-value before any assignment is done.

The mapping from the L-value to its previous R-value is changed by the execution of an assignment statement, such that the L-value subsequently contains the new R-value.

The return value of an assignment statement is always **dummy**. I.e. it is only useful for the "side-effect" of it evaluation.

The discarded R-value will be automatically detstroyed by the GTL garbage-collector which runs behind the scenes. The memory occupied by the discarded R-value will be recycled for re-use.

Note: R-values are not "shared" only the L-values are shared between variables declarations an/or tuple elements – so the discarded R-value from an assignment statement will always be garbage-collected. (although if the discarded R-value is a tuple, some of the tuples elements may not be garbage collected if they are shared with other variables or tuples).

Example 1

```
\demonstration of a simple while statement
let i = 10 in                \declare a variable i and initialise it to be 10
while (i > 0) do             \while i is greater than 0
{     i; " ";                \print out the value of i
      i := i - 1             \decrement the value of i
}
```

Output to Example 1

10 9 8 7 6 5 4 3 2 1

Example 2

```
let i, x, y = 0, 1, 1 in
i, x, y := x, y, i;
i; x; y
```

Output to Example 2

110

Addition Assignment Statement

Syntax

E_1 += E_2

Description

E_1 is evaluated to yield an L-value and E_2 is evaluated to yield an R-value before any assignment is done.

The mapping from the L-value to its previous R-value is changed by the execution of an assignment statement, such that the L-value subsequently contains the new R-value.

The return value of an assignment statement is always **dummy**. I.E. it is only useful for the “side-effect” of it evaluation.

The discarded R-value will be automatically detstroyed by the GTL garbage-collector which runs behind the scenes. The memory occupied by the discarded R-value will be recycled for re-use.

Note: R-values are not “shared” only the L-values are shared between variables declarations an/or tuple elements – so the discarded R-value from an assignment statement will always be garbage-collected. (although if the discarded R-value is a tuple, some of the tuples elements may not be garbage collected if they are shared with other variables or tuples).

Example 1

```
\demonstration of a simple while statement
let i = 0 in                  \declare a variable i and initialise it to be 0
while (i < 10) do             \while i is less than 10
{     i; " ";                 \print out the value of i
      i += 1                  \increment the value of i
}
```

Output to Example 1

0 1 2 3 4 5 6 7 8 9

Comma Operator

Syntax

T,*E*

Description

The comma is an infixed, non-associative tuple maker. It is the comma that makes the tuple, not any bracketing, unless GTL's grammatical rules would otherwise indicate an alternative grouping.

Example 1

```
1,2,3;          \a tuple of order 3, each element is a REF
(1,2),3;        \a tuple of order 2, the first element is itself a tuple of order 2
1,(2,3)         \a tuple of order 2, the 2nd element is itself a tuple of order 2
```

Output to Example 1

(1, 2, 3)((1, 2), 3)(1, (2, 3))

Notes:
The , operator is often used in multiple declarations and multiple assignments. E.g:

```
Let x, y, z = 1, 2, 3 in . . .

UserName, Password, ExpiryDate := "GEORGE", "QUERTY", today();
```

In the assignment statement example the left hand side is evaluated to create a tuple of L-values, and the right hand side expression is evaluated to create a tuple of R-values.

Aug Operator

Syntax

T1 **aug** *E*

Description

aug is a tuple making operator, whose left operand must be a tuple *T1* of order N and whose right operand maybe any expression *E*. The resultant tuple is of order N+1.

Example 1

```
let t = (1, 2, 3) in
let e = "test" in
t aug e
```

Output to Example 1

(1, 2, 3, test)

Example 2

```
let t = (1,2,3) in
let e = (1,2) in
t aug e
```

Output to Example 2

(1,2,3,(1,2))

Note that the implementation of the **aug** operator in GTL is philosophically flawed, in comparison with the original PAL language concept. The following example:

```
let t = (1, 2) in
let s = t aug 3 in
t
```

will produce output (1, 2, 3)

because the GTL interpreter implements **aug** by modifying the original tuple R-value, not by copying it an then augmenting it.

It is advisable therefore, to restrict the usage of **aug** to the same r-value for example always use it in expressions of the form:

t := t aug x

This compromise has been made because tuples are often large in typical GTL applications and the performance penalty of copying the whole tuple every time an aug is executed would be unacceptable.

An alternative operator has been implemented using **au** as the token which is the “pure” version, in that it makes a copy of the original tuple argument. The l-values from the original tuple will be still be shared by the new tuple.

Logical Or Operator

Syntax

b1 **or** *b2*

Description

The Boolean expression *b1* is evaluated first. If value of *b1* is zero *b2* is evaluated and it's value returned. If *b1* is non-zero, it's value is returned, and *b2* is never evaluated.

Example 1

```
let t = () in
if (null t or t 1 = "a") do   \if t 1 = "a" was evaluated in this instance an
                              \error would occur
                              \but it is not evaluated because null t is true
t := t aug 1;
t
```

Output to Example 1

(1)

Bit-wise Or Operator

Syntax

e1 || *e2*

Description

Both expressions are evaluated and there values are or-ed together as bit-patterns and the result returned as the value of the sub-expression. Note this convention is back to front from C Language usage where the single | operator is the bit-wise operator, and || is the logical operator.

Logical And Operator

Syntax

b1 **&** *b2*

Description

The Boolean expression *b1* is evaluated first. If the value of *b1* is non-zero *b2* is evaluated and it's value yielded as the value of the sub-expression. If the value *b1* is zero, zero is yielded, as the value of this sub-expression and *b2* is never evaluated.

Example 1

```
let t = () in
t := t aug "a";
if (not(null t) & (t 0 = "a")) do
      t := t aug 1;
t
```

Output to Example 1

(a, 1)

Bit-wise And Operator

Syntax

e1 && e2

Description

Both expressions are evaluated and there values are and-ed together as bit-patterns and the result returned as the value of the sub-expression. Note this convention is back to front from C Language usage where the single & operator is the bit-wise operator, and && is the logical operator,

Not Expression

Syntax

not *e1*

~ *e1*

Description

This is not a bit wise operator. If the value of *e1* is zero the value of not is 1. If the value of *e1* is non-zero, the value of not is zero.

Example 1

```
not 1
```

Output to Example 1

0

Example 2

not 2

Output to Example 2

0

~~ Expression

The double tilde operator implements a bit wise ones complement operator. e.g. ~~ 0 yields –1, and ~~ 1 yields -2 that is to say 31 binary ones and a low order binary 0.

Addition Expression

Syntax

E1 + *E2*

Description

This is the arithmetic addition operator. The result is the sum of the value of expression *E1* and the value of expression *E2*. A *ref* value and a *num* value can be added together and the result is a *num* value. Where *E1* **and** *E2* are two conforming vectors (tuples), vector addition is performed. If one is scalar and one a vector scalar addition is performed.

Example 1

```
1 + 2
```

Output to Example 1

3

Example 2

```
1 + 2.1
```

Output to Example 2

3.100000

Minus Expression

Syntax

r1 – r2

– r2

Description

This is the arithmetic subtraction operator. The result is the subtraction of expression *E2* from expression *E1*. A *ref* value and a *num* value can be subtracted and the result is a *num* value. . Where *E1* **and** *E2* are two conforming vectors (tuples), vector subtraction is performed. If one is scalar and one a vector scalar subtraction is performed.

Example 1

```
2.3 - 1
```

Output to Example 1

1.300000

Example 2

```
-(-1)
```

Output to Example 2

1

Division Expression

Syntax

r1 / r2

Description

This is the arithmetic division operator. The result is the division of the value of expression E_1 by the value of expression E_2. A *ref* value divided by a *ref* value will yield a *ref* result with the remainder discarded. Mixed *ref* and a *num* values will yield a *num* result.

Division by zero or a very small number will cause a *GTL Execution Time Error.*

It is wise to test the value of the divisor unless the programmer is certain that it can never be zero.

The **sig** operator is available e.g. **sig(6, y)** is true if y is significant to six decimal places.

Example 1

```
1/2
```

Output to Example 1

0

Example 2

```
1/2.0
```

Output to Example 2

0.500000

Remainder Expression

Syntax

r1 **rem** *r2*

Description

This is the arithmetic remainder operator. Both value must be if type ref. The value of the expression is the remainder after division of *r1* by *r2*.

Example 1

```
3 rem 2
```

Output to Example 1

1

Multiplication Expression

Syntax

*r1 * r2*

Description

This is the arithmetic multiplication operator. The result is the product of the values yielded by expression E_1 and expression E_2. A *ref* value times a *ref* value yields a *ref* result. A *num* times a *ref* or *ref* times a *num* yields a *num* result.

Example 1

```
2 * 3
```

Output to Example 1

6

Dot Product Operator

```
{       let a = 1, 2, 3 in
        let b = 3. 4. 5
        a · b
}
```

Will output the scalar dot product of two vectors: 26

Cross Product Operator

```
{       let a = 1, 2, 3 in
        let b = 3. 4. 5
        a × b
}
```

Will output the cross product of two vectors: (-2, 4, -2)

Power Operator

Syntax

*r1**r2*

Description

The power operator calculates *r1* raised to the power of *r2*.

If *r2* < 0 the return value will be 1.0.

If *r1* < > 0 and *r2* = 0.0 the return value will be 1, and if *r1* = 0.0 and *r2* = 0.0 the return value will be 1.0.

The arguments may be mixed ref or num. The result is always num.

Example 1

```
2**3 will output 8.000000
```

Example 2

```
4**(1.0/2.0) will output 2.000000
(note there is no sqrt operator in GTL so the x**0.5 is a good trick).
```

Concatenation Operator

Syntax

s1 . s2

Description

The concatenation operator appends *s2* to *s1* and returns the result as a new string.

This is certainly the most frequently exploited operator by GTL programmers. The management of string values in automatically garbage collected memory is an immensely powerful concept compared to most - more primative programming environments.

To labour the point you can throw a string away in GTL whether it is 10 bytes or 10 million bytes long with absolutely no penalty in terms of memory leakage or execution delay.

Example 1

```
let s = "this is a " in
let s2 = "test!" in
s. s2
```

Output to Example 1

this is a test!

Example 2

```
let s = "this is a " in
let s2 = "test!" in
let NL = "
" in
s. s2; NL;
s
```

Output to Example 2

this is a test!
this is a

Sub-string Operator

Syntax

s1`!(`*e1*`, `*e2*`)`

The exclamation mark operator is an infix operators which returns a sub-string of its first argument, determined by the *e1* and *e2* arguments. *e1* is the starting offset for the sub-string, and *e2* is the length of the sub-string. *e1* must be less than the length of the *s1* string, but *e2* can be any value.

Expression of the form **`Postcode!(0, 10)`** are a convenient and efficient way of guaranteeing a fixed string length when required.

Example 1

```
"The quick brown fox"!(4, 5)
```

Output from Example 1

```
quick
```

Example 2

```
("Fred Smith"!(0, 16)). "x"
```

Output from Example 1

```
Fred Smith      x
```

Relational Operators

Syntax

e1 **<** *e2*
e1 **>** *e2*
e1 **<=** *e2*
e1 **=>** *e2*
e1 **=** *e2*
e1 **<>** *e2*

Description

Expressions *e1* and *e2* must be of the same type, and must be either a *ref*, *num*, *string* or *tuple* value.

The relational equality operators compare *e1* to *e2* to test the validity of the specified relationship. The result of a relational expression is not 0 if the tested relationship is true and 0 if it is false. The value returned is always a *ref* value.

When tuples are compared they are compared element by element and recursively for sub-tuples.

Example 1

```
1 < 0
```

Output to Example 1

0

Example 2

```
"test" < "tes"
```

Output to Example 2

0

Alternative versions of the form **%<** **%<=** **%>** **%>=** are available to provide unsigned 32-bit ref comparisons for comparing 32-bit values generally employed as memory or disk addresses.

Bracket Operators

Syntax

(*e1***)**
[*e1***]**
{*e1***}**

Description

You can enclose *e1* in parentheses without changing the type or value of the enclosed expression.
In GTL all bracketing is considered to be the same. There is no distinction between **{**, **[**, and **(**. It is however conventional to use **{}** to define scope, and **()** for evaluation of expression e1, with **[]** rarely used.

Example 1

```
let sqrt x = valof                  \define the function
{     res x**1/2                    \correct answer for wrong reason
} in
sqrt 4
```

Output to Example 1

2.000000

Example 2

```
let sqrt x = valof                  \define the function
{     res x**(1/2)                  \in this line 1/2 evaluates to 0
} in
sqrt 4
```

Output to Example 2

1.000000

Example 3

```
let sqrt x = valof                  \define the function
{     res x**(1.0/2.0)              \correct!
} in
sqrt 4
```

Output to Example 2

2.000000

Example 1 is wrong because x is raised to power of one and then divided by 2.

Example 2 is wrong even though the brackets are right because integer 1 divide by integer 2 gives 0.

Example 3 is correct as (1.0/2.0) evaluates to 0.5

For Operator

Syntax

for *d* **to** *e1* **by** *e2* **in** *e3*
for *d* **to** *e1* **in** *e3*

Description

Definition *d* can be a single scalar variable, a list of scalar variables, or a function/procedure.

If *d* is to define a single scalar variable, *d* must be have the syntax n_1 **=** e_0 where n_1 will then be initialised to the value of e_0. n_1 must be either a *num*, or *ref* value. Expressions *e1* must evaluate to *ref* or *num* values. n_1, *e1*, *e2* must all be the same type. *e1* and *e2* are evaluated only once, and this is before *e3* is ever evaluated. If the **by** statement is missing *e2* is assumed to be 1.

Note that while **for** is generally similar to **let** in that it is a (re-)declaration of the variable, there is an important difference in that an implied “unshare” operator is invoked such that a new l-value is always created.

The extent of the definition *d* is confined to the expression *e3*.

Example 1

```
let NL = "
" in
for i = 9 to 0 by -1 in
{     i;
      for i = 0 to 9 in
            i;
      NL
}
```

Output to Example 1

90123456879
80123456789
70123456789
60123456789
50123456789
40123456789
30123456789
20123456789
10123456789
00123456789

foreach Operator

Syntax

`foreach` *d* **`of`** *e1* **`in`** *e3*

Description

Definition *d* can be a single scalar variable, or a list of scalar variables.

e1 is a tuple of conforming elements. If d is a single scalar variable then members of e1 can be any values, scalars or tuples. If d is a list of variable names, then the members of e1 must consist of tuples all of the same order as d.

The extent of the definition *d* is confined to the expression *e3*.

Expression e3 is evaluated once for each member of e1, with the variables defined in d bound to the members of e1.

Inside the evaluation of e3 a "hidden" pre-defined local variable with identifier _i is available which starts with a value of 0 and increments to n-1 for each of the elements of e1.

The **foreach** operator causes an output event for each iteration i.e. it is imperative in form.

forall Operator

Syntax

`forall` *d* **`of`** *e1* **`in`** *e3*

Description

Generally similar to the foreach operator except **forall** is applicative in context. It returns a tuple where each element is the result of one evaluation of e3

str Operator

Syntax

`str` *d* **`of`** *e1* **`in`** *e3*

Description

Definition *d* can be a single scalar variable, or a list of scalar variables.

e1 is a tuple of conforming elements. If d is a single scalar variable then members of e1 can be any values, scalars or tuples. If d is a list of variable names, then the members of e1 must consist of tuples all of the same order as d.

The extent of the definition *d* is confined to the expression *e3*.

Expression e3 is evaluated once for each member of e1, with the variables defined in d bound to the members of e1.

Inside the evaluation of e3 a "hidden" pre-defined local variable with identifier _i is available which starts with a value of 0 and increments to n-1 for each of the elements of e1.

The **str** operator is intended for an expression context. It expects each iteration to yield a string value and returns a concatenated string with all iterations combined.

Unshare Operator

Syntax

$e1

Description

The unshare operator returns a unique l-value. This operator is only meaningful in a context where an L-value is expected e.g. in a declaration, or in the creation of a tuple.

The unshare concept is very important – a full understanding of the implications of sharing is crucially important to the successful GTL programmer.

Example 1

```
\program using unshare operator
let NL = "
" in
let x = 1 in
let y = $x in
x; y;
x := 2;
x; y;
y := 3;
x; y
```

Output to Example 1

11
21
23

Example 2

```
\program not using unshare operator
let NL = "
" in
let x = 1 in
let y = x in
x; y;
x := 2;
x; y;
y := 3;
x; y
```

Output to Example 2

11
22
33

Sharing in GTL needs to be well understood by the programmer. Especially in relation to the use of tuples.
For example:

```
let a, b = "Michael", "Doug" in
let Names = (a, b) in
b := "George";
Names; NL
```

Will output

(Michael, George)

Nil Operator

Syntax

```
nil
```

Description

Returns a tuple of order zero.

Example 1

```
nil
```

Output to Example 1

()

Quote Operators

Syntax

```
"str"
```

Description

Double quote characters are used to delimit a literal string. The quotes can appear over several lines. The value created is stored in managed memory and may be discarded at any time without leakage issues. There are several operators in GTL available to manipulate strings. The period (dot) character is the concatenation operator , the ! operator is the sub-string operator, *stem, stern, last, front* are available to pick strings apart, and applying a string to a ref value returns a particular character at a 0 based offset from the start of the string.

String handling in GTL is extremely powerful and provides excellent ability to create and manage all sorts of textual data including ASCII, HTML, XML, EDI etc, etc.

Example 1

```
"This
is
a
test"
```

Output to Example 1

This
is
a
test

switch Operator

Syntax

switch (*e1*, *e2*, *t***)**

Description

This is an applicative switch concept generally like the “C” switch operator.

All three expressions are evaluated to yield a 3-tuple.

The value of *e1* must be a ref integer in the range 0 to n-1 where n is the order of the tuple which is the value of t above. *t* must evaluate to a tuple of values which are applicable to the value of *e2*. i.e. if *t* is a tuple of lambda expressions the number of bound variables must agree with the order of the value of *e2*. If *t* is a tuple of strings or tuples, then *e2* must be a ref integer.

Example 1

```
switch (3,
            (“Arg1”, “Arg2”, Arg3”),
                  (fn(x, y, z)(“Case 0 “.x.”,”.y.”,”.z))),
                  (fn(x, y, z)(“Case 1 “.x.”,”.y.”,”.z))),
                  (fn(x, y, z)(“Case 2 “.x.”,”.y.”,”.z))),
                  (fn(x, y, z)(“Case 3 “.x.”,”.y.”,”.z))),
                  (fn(x, y, z)(“Case 4 “.x.”,”.y.”,”.z))),
                  (fn(x, y, z)(“Case 5 “.x.”,”.y.”,”.z)))
            )
         )
```

Output to Example 1

```
case 3 Arg1, Arg2, Arg3
```

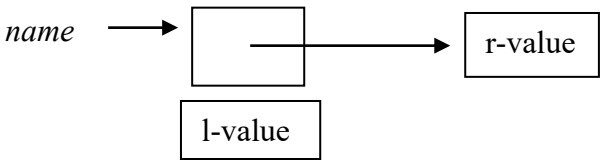
has & same Operators

syntax

`T` **`has`** *`e`*

`e1` **`same`** *`e2`*

These operators relate to the fundamental ontological basis for variables and values in the GTL Language.



A variable name is bound to an l-value which may be thought of as a container for an r-value. R-values consist of scalar elements such as ref integers, num floating point values, strings and non-scalars such as tuples and objects.

The terms l-value and r-value derive from their roles on the lhs and the rhs of an assigment operator **:=**

The assignment operator changes the connection betweenan l-value and an r-value.

The connection between a variable name and an l-value never changes, though a sinlge l-value may be connected to more than one name.

Tuples are ordered sets of anonymous l-values each of which may be conncted to one or more vraiable names.

```
let Siblings = ("Mary", "George", "Sally")  in
let Sisters = (Siblings 0, Siblings 2) in
let Brother = Siblings 1 in
```

So the l-value containing the r-value "George" only exists once but has two connections to it. *Brother* & *Siblings 1*

The **has** operator tells you if an l–value is present in a tuple so

`Siblings` **`has`** `Brother` evaluates to 2

`Sisters` **`has`** `Brother` evaluates to 0

The same operator tells you if two l-values are identical

`Brother` **`same`** `(Siblings 1)` evaluates to 1

`Brother` **`same`** `(Sisters 1)` evaluates to 0

Note the value returned by the **has** operator is the offeset in the tuple (1 to n) or 0 if the l-value is not found.

The **same** operator returns 0 ro 1.

Note: the **lv** operator is available as a diagostic aid about sharing. It returns a ref value unquely identifying the l-value.

GTL Syntax

GTL Non-terminal Symbols

Label	Type
P	Program
E	Expression
E1	where expression
E2	valof expression
C	command
C1	labelled command
C2	conditional command
C3	basic command
T	tuple
T1	non null tuple
T2	conditional expression
B	boolean
B1	conjuction
B2	negation
B3	relation
A	arithmetic expression
A1	multiplication / division expression
A2	factor expression
A3	primary expression
OP	operation
R	combination
R1	rand
D	definition
D1	simultaneous definition
D2	rec definition
D3	basic definition
V	bv part
V1	basic bv
NL	NAME list
RL	relational functor
NAME	variable identifier

GTL Syntax

This is the GTL syntax in Bacchus Naur form

```
P:          E
       ;

E:          _let D _in E
       |    _fn V1 _dot E
       |    E1
       ;

E1:         E2 _where D2
       |    E2
       ;

E2:         _valof C
       |    C
       ;

C:          C1 _semicolon C
       |    C1
       ;

C1:         _if B _do C1 _else C1
       |    _if B _do C1
       |    _for D _to E _by E _in C1
       |    _for D _to E _in C1
       |    _unless B _do C1
       |    _while B _do C1
       |    C3
       ;

C3:         T _assign T
       |    T _plusequal T
       |    _res T
       |    _checkvar T
       |    T
       ;

T:          T1
       |    T _comma T1
       ;

T1:         T1 _aug T2
       |    T2
       ;

T2:         B _cond T2 _bar T2
       |    B
       ;

B:          B _logor B1
       |    B _bitor B1
       |    B1
       ;

B1:         B1 _logand B2
       |    B1 _bitand B2
       |    B2
       ;

B2:         _not B3
       |    _bitnot B3
       |    B3
       ;

B3:         A RL A
       |    A
       ;
```

```
A:          A _plus A1
      |     A _dot A1
      |     A _colon A1
      |     A _exclamation A1
      |     A _minus A1
      |     _minus A1
      |     A1
      ;

A1:         A1 _times A2
      |     A1 _divide A2
      |     A1 _rem A2
      |     A1 _lshift A2
      |     A1 _rshift A2
      |     A2
      ;

A2:         A3 _power A2
      |     A3
      ;

A3:         OP
      |     _unshare OP
      ;

OP:   <see main manual above for a list of pre-defined operators>
      |     R
      ;

R:          R R1
      |     R1
      ;

R1:         _reflit
      |     _numlit
      |     QUOTATION
      |     _ident
      |     _nil
      |      _bra E _ket
      |     _sqbra E _sqket
      |     _curb E _cket
      ;

D:          D1
      ;

D1:         D1 _and D2
      |     D2
      ;

D2:   _rec D3
      | D3
      ;

D3:         NL _eq E
      |     NAME V _eq E
      |     _bra D _ket
      |     _sqbra D _sqket
      ;

V:          V V1
      |     V1
      ;

V1:         _bra NL _ket
      |     _nil
      |     NAME
      ;

NL:         NL _comma NAME
```

```
        |       NAME
        ;

RL:             _ls
        |       _gt
        |       _le
        |       _ge
        |       _eq
        |       _ne
        ;

NAME:           _ident
        ;
```