The PEBL Manual

Shane T. Mueller

Current for PEBL Version 0.09 – August 2008
http://pebl.sourceforge.net

Contents

Chapter 1

About

PEBL (Psychology Experiment Building Language) is a cross-platform, open-source programming language and execution environment for constructing programs to conduct a wide range of archetypal psychology experiments. It is entirely free of charge, and may be modified to suit your needs as long as you follow the terms of the GPL, under which the source code is licensed. PEBL is written primarily in `C++`, but requires a few other tools (`flex`, `yacc`) and libraries (`SDL`, `SDL_image`, `SDL_gfx`, and `SDL_ttf`) to use. It currently compiles and runs on Linux (using `g++`), Mac OSX (also using `g++`), and Microsoft Windows (using `Dev-cpp` and `mingw`) platforms using free tools. It has been developed primarily by Shane T. Mueller, Ph.D. (smueller@obereed.net). This document was prepared with editorial and formatting help from Gulab Parab and Samuele Carcagno. Contributions are welcome and encouraged.

Chapter 2

Usage

2.1 How to Compile

Currently, there is no automated compile procedure. PEBL requires the `SDL`, `SDL-image`, `SDL-gfx` and `SDL-ttf` libraries and development headers. It also uses `flex` and `bison`, but you can compile without these tools. PEBL compiles on both Linux and Windows using the free `gcc` compiler. Note that `SDL-image` may require `jpeg`, `png`, and a `zlib` compression library, while `SDL-ttf` requires `truetype 2.0`.

2.1.1 Linux

PEBL should compile by typing '`make`' in its base directory once all requisite tools are installed and the source distribution is uncompressed. Currently, PEBL does not use autotools, so its make system is rather brittle. Assistance is welcome.

On Linux, compiling will fail if you don't have an `/obj` directory and all the appropriate subdirectories (that mirror the main tree.) These will not exist if you check out from CVS.

2.1.2 Microsoft Windows

On Microsoft Windows, PEBL is designed to be compiled using the Free ide `dev-c++` available at http://www.bloodshed.net/dev/devcpp.html and instructions for installing `dev-c++`, `sdl`, and the `minGW` system can be found at:

http://www.libsdl.org/pipermail/sdl/2002-June/046382.html

and elsewhere on the net. Email the PEBL list for more details.

2.1.3 Mac OSX

Currently, (versions after 0.09) PEBL in untested on Macintosh. It should compile and run acceptably, but we do not currently distribute a binary version.

PEBL is compiled and runs from the command-line, and resides in `/usr/local/share/pebl` and `/usr/local/bin`. To compile, download the source and compile `libSDL`, `libSDL_image`, and `libSDL_ttf`. `libSDL_ttf` requires freetype, and `libSDL_image` requires `libpng`, `libjpeg`, and `zlib`.

2.1.4 Linux

On Linux, you will probably have to install from source. There is an '`install`' option in the Makefile that, if invoked with '`make install`', will copy `bin/pebl` to `/usr/local/bin/pebl`, and the `media/` directories to `/usr/local/share/pebl/media`. You must be root to do this, and you can just as easily do it by hand.

2.1.5 Microsoft Windows

In Microsoft Windows, we provide an installer package that contains all necessary executable binary files and `.dlls`. This installer places PEBL in `c:\Program Files\PEBL`, and creates a directory `pebl-exp` in `My Documents` with a shortcut that allows PEBL to be launched and programs that reside there to be run.

2.1.6 Macintosh OSX

For the MAC, we provide a `.pkg` installer that installs all the necessary files and libraries in `/usr/local/share/pebl`. PEBL can be run from the command line by invoking `/usr/local/bin/pebl`.

2.2 How to Run a PEBL Program

2.2.1 Linux

If you have installed PEBL into `/usr/local/bin`, you will be able to invoke PEBL by typing '`pebl`' at a command line. PEBL requires you to specify one or more source files that it will compile and run, e.g., the command:

```
> pebl stroop.pbl library.pbl
```

will load the experiment described in `stroop.pbl`, and will load the supplementary library functions in `library.pbl`.

Additionally, PEBL can take the `-v` or `-V` command-line parameter, which allows you to pass values into the script. This is useful for entering subject numbers and condition types using an outside program like a bash script (possibly one that invokes dialog or zenity). A sample zenity script that asks for subject number and then runs a sample experiment which uses that input resides in the

`bin` directory. The script can be edited to use fullscreen mode or change the display dimensions, for example. See Section 2.4: Command-Line Arguments.

You can also specify directories without a filename on the command-line (as long as they end with '`/`'). Doing so will add that directory to the search path when files are opened.

2.2.2 Microsoft Windows

PEBL can be launched from the command line in Windows by going to the `pebl\bin` directory and typing '`pebl.exe`'. PEBL requires you to specify one or more source files that it will compile and run. For example, the command

```
> pebl stroop.pbl library.pbl
```

loads the experiment described in `stroop.pbl`, and loads supplementary library functions in `library.pbl`.

Additionally, PEBL can take the `-v` or `-V` command-line parameter, which allows you to pass values in to the script. This is useful for entering condition types using an outside program like a batch file. the `-s` and `-S` allow one to specify a subject code, which gets bound to the gSubNum variable. If no value is specified, gSubNum is initialized to 0. You can also specify directories without a file (as long as they end with '\'). Doing so will add that directory to the search path when files are opened. See Section 2.4: Command-Line Arguments.

Launching programs from the command-line on Windows is cumbersome. One easy way to launch PEBL on Windows is to create a shortcut to the executable file and then edit the properties so that the shortcut launches PEBL with the proper script and command-line parameters. Another way is to write and launch a batch file, which is especially useful if you wish to enter configuration data before loading the script.

Win32 Launcher

PEBL comes with a launcher program that launches PEBL scripts in Microsoft Windows (tm). It will allows you to specify variables to pass into PEBL on execution, select multiple source files to load, and configure with a text file.

The launcher is written in Visual Basic, and so you might need some `.dll` files in order for it to run.

When the launcher is run, it first looks for a file called `pebl-init.txt`. This file should have the following format:

```
---------------Beginning of file----------------------
"Quoted_Path_To_PEBL_Executable"
"Quoted_Path_To_Directory_To_Load_Files_From"
"File1.pbl" "File2.pbl"
1st Variable Label|Initial_Value
2nd Variable Label|Initial_Value
3rd Variable Label|Initial_Value
```

```
.
.
.
------------------End of file-------------------------
```

My actual pebl-init looks like this:

```
-------------
"c:\Documents and Settings\smueller\My Documents\pebl\bin\pebl.exe"
"c:\Documents and Settings\smueller\My Documents\pebl\demo\"
"hello.pbl"
Subject Number|1
Condition|fast
-------------
```

The Launcher will select the specified files in the listed directory (this can be changed by selecting other files). After the third line, every pair separated with a '|' will appear as a text-entry box with the pre-specified default value. This can be used to specify subject numbers, conditions, and such, which are then fed into the PEBL script.

As of version 0.06, the launcher is improved so that it will open the `stdout.txt` and `stderr.txt` files after a script has been run and display them in a tabbed interface at the bottom of the launcher window.

2.2.3 Macintosh OSX

The latest version of PEBL packaged for OSX is 0.07. Until hardware/developers are available, newer versions for OSX will need to be compiled on your own.

Installing `pebl.pkg` places PEBL in `/usr/local/share/pebl` and `/usr/local/bin`. Currently, PEBL must be run from the command-line (there is no graphical front-end). Open a terminal (in the applications folder) and type at the `$` prompt:

```
$ /usr/local/bin/pebl
```

To execute, type:

```
$  /usr/local/bin/pebl Documents/test.pbl
```

To truncate, add `/usr/local/bin to the path`:

```
export PATH=$PATH:/usr/local/bin
```

Then you can run:

```
$  pebl Documents/test.pbl
```

On OSX, there is no such thing as double-buffering. However, under fullscreen mode, drawing can be synced to the vertical refresh. But as a caveat, this has not yet been implemented.

2.3 How to stop running a program

In order to improve performance, PEBL runs at the highest priority possible on your computer. This means that if it gets stuck somewhere, you may have difficulty terminating the process. We have added an 'abort program' shortcut key combination that will immediately terminate the program and report the location at which it became stuck in your code:
press `<CTRL><SHIFT><ALT><\>` simultaneously.

2.4 Command-line arguments

Some aspects of PEBL's display can be controlled via command-line arguments. Some of these are platform specific, or their use depends on your exact hardware and software. The following guide to command-line arguments is adapted from the output produced by invoking PEBL with no arguments:

Usage: Invoke PEBL with the experiment script files (`.pbl`) and command-line arguments.

Examples:

```
pebl experiment.pbl -s sub1 --fullscreen --display 800x600 --driver dga
pebl experiment.pbl --driver xf86
pebl experiment.pbl -v 33 -v 2 --fullscreen --display 640x480
```

Command-Line Options

`-v VALUE1 -v VALUE2`

Invokes script and passes `VALUE1` and `VALUE2` (or any text immediately following a `-v`) to a list in the argument of the `Start()` function.
This is useful for passing in conditions, subject numbers, randomization cues, and other entities that are easier to control from outside the script. Variables appear as strings, so numeric values need to be converted to be used as numbers.

`-s VALUE`
`-S VALUE`

Binds `VALUE` to the global variable gSubNum, which is set by default to 0.

`--driver <drivername>`

Sets the video driver, when there is more than one. In Linux SDL, options `xf86`, `dga`, `svgalib` (from console), itcan also be controlled via environment variables. In fact, for SDL versions of PEBL simply set the `SDL_VIDEO_DRIVER` environment variable to the passed-in argument, without doing any checking, and without checking or returning it to its original state.

`--display <widthxheight>`

Controls the screen width and height (in pixels). Defaults to `640x480`. Currently, only the following screens are supported:

`512x384`

`640x480`

`800x600`

`960x720`

`1024x768`

`1152x864`

`1280x1024`

Note: the way this is invoked may be changed in the future. Your video display may not support the command-line argument. If it does not, PEBL should exit and disply a useful error message; of course, it could possibly damage your hardware.

For the sake of convenience, the width, height, and bit depth can be accessed from within a PEBL script using the global variables `gVideoWidth`, `gVideoHeight`, and `gVideoDepth`. If these values are set within a script before the function `MakeWindow()` is called, the window will be created with these values, overriding any command-line parameters.

`--depth`

Controls the pixel depth, which also depends on your video card. Currently, depths of 2,8,15,16,24, and 32 are allowed on the command-line. There is no guarantee that you will get the specified bit depth, and bit depths such as 2 and 8 are likely never useful. Changing depths can, for some drivers and video cards, enable better performance or possibly better video sychrony.

`--windowed` or `--fullscreen`

Controls whether the script will run in a window or fullscreen.

Chapter 3

How to Write a PEBL Program

3.1 Basic PEBL Scripts

PEBL has a fairly straightforward and forgiving syntax, and implements most of its interesting functionality in a large object system and function library of over 125 functions. The library includes many functions specific to creating and presenting stimuli and collecting responses. Efforts, however successful, have been made to enable timing accuracy at amillisecond-scale, and to make machine limitations easy to deal with.

Each PEBL program is stored in a text file. Currently, no special authoring environment is available. A program consists of one or more functions, and *must* have a function called `Start()`. Functions are defined with the following syntax:

```
define <function_name>(parameters)
{
  statement 1
  statement 2
  ....
  return value3
}
```

The parameter list and the return value are optional. For the `Start(par){}` function, `par` is normally bound to 0. However, if PEBL is invoked with `-v` command-line parameters, each value that follows a `-v` is added to a list contained in '`par`', which can then be accessed within the program:

```
define Start(par)
{
   Print(First(par))
}
```

A simple PEBL program that actually runs follows:

```
define Start(par)
{
 Print("Hello")
}
```

`Print()` is a standard library function. If you run PEBL from a command-line, the text inside the `Print` function will be sent to the console. On Windows, it will appear in the file '`stdout.txt`' in the PEBL directory. Although other functions do not need a parameter argument, the `Start()` function does (case values are passed in from the command-line).

A number of sample PEBL programs can be found in the `/demo` subdirectory.

3.2 Case Sensitivity

PEBL uses case to specify an item's token type. This serves as an extra contextual cue to the programmer, so that the program reads more easily and communicates more clearly.

Function names must start with an uppercase letter, but are otherwise caseinsensitive. Thus, if you name a function `"DoTrial"`, you can call it later as `"DOTRIAL"` or `"Dotrial"` or even `"DotRail"`. We recommend consistency, as it helps manage larger programs more easily.

Unlike function names, variable names must start with an lowercase letter; if this letter is a 'g', the variable is global. This enforces a consistent and readable style. After the first character, variable names are caseinsensitive. Thus, the variable '`mytrial`' is the same as '`myTrial`'.

Currently, syntax keywords (like loop, if, define, etc.) must be lowercase, for technical reasons. We hope to eliminate this limitation in the future.

3.3 Syntax

PEBL has a simple and forgiving syntax, reminiscent of `S+` (or `R`) and `c`. However, differences do exist.

Table 3.1 shows a number of keywords and symbols used in PEBL. These need not appear in lowercase in your program.

Note that the '`=`' symbol does not exist in PEBL. Unlike other languages, PEBL does not use it as an assignment operator. Instead, it uses '`<-`'. Because it is confusing for users to keep track of the various uses of the `=` and `==` symbols, we've eliminated the '`=`' symbol entirely. Programmers familiar with `c` will notice a resemblance between PEBL and `c`. Unlike `c`, in PEBL a semicolon is not necessary to finish a statement. A carriage return indicates a statement is complete, if the current line forms a complete expression. You may terminate every command with a '`;`' if you choose, but it may slow down parsing and execution.

Table 3.1: PEBL Symbols and Keywords

Symbol/Keyword	Usage
`+`	Adds two expressions together
`-`	Subtracts one expression from another
`/`	Divides one expression by another
`*`	Multiplies two expressions together
`^`	Raises one expression to the power of another
`;`	Finishes a statement, or starts a new statement on the same line (is not needed at end of line)
`.`	The property accessor. Allows properties to be accessed by name
`<-`	The assignment operator
`( )`	Groups mathematical operations
`{ }`	Groups a series of statements
`[ ]`	Creates a list
`#`	Comment—ignore everything on the line that follows
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equal to
`<> != ~=`	Not equal to
`and`	Logical and
`break`	Breaks out of a loop prematurely
`not`	Logical not
`or`	Logical or
`while`	Traditional while loop
`loop`	Loops over elements in a list
`if`	Simple conditional test
`if...else`	Complex conditional test
`if...elseif...else`	Extended conditional chain
`define`	Defines a function
`return`	Allows a function to return a value

Another difference between `c` and PEBL is that in PEBL, `{}` brackets are not optional: they are required to define code blocks, such as those found in `if` and `while` statements and loops.

3.4 Expressions

An expression is a set of operations that produces a result. In PEBL, every function is an expression, as is any single number. Expressions include:

```
3 + 32
(324 / 324) - Log(32)
not lVariable
Print(32323)
"String " + 33
nsuho  #this is legal if nsuho has been defined already.
```

Notice that `"String "  33+` is a legal expresison. It will produce another string: `"String 33"`.

These are not expressions:

```
NSUHO         #Not an expression
( 33 + 33     #Not an expression
444 / 3342 +  #Not an expression
```

`NSUHO` is not a variable because it starts with a capital letter. The other lines are incomplete expressions. If the PEBL parser comes to the end of a line with an incomplete expression, it will automatically go to the next line:

```
Print("hello " +
       " world."
     )
```

This can result in bugs that are hard to diagnose:

```
a <- 33 + 323 +
Print(1331)
```

sets `a` to the string `"3561331"`.

But if a carriage return occurs at a point where the line does make a valid expression, it will treat that line as a complete statement:

```
a <- 33 + 323
 * 34245
```

sets `a` equal to 356, but creates a syntax error on the next line.

Any expression can be used as the argument of a function, but a function may not successfully operate when given bogus arguments.

If a string is defined across line breaks, the string definition will contain a linebreak character, which will get printed in output text files and textboxes.

```
text <- "this is a line
and so is this"
```

If you desire a long body of text without linebreaks, you must define it piecemeal:

```
text <- "This is a line " +
        "There is no line break before this line."
```

3.5 Variables

PEBL can store the results of expressions in named variables. Unlike many programming languages, PEBL only has one type of variable: a "Variant". This variable type can hold strings, integers, floating-point numbers, lists, graphical objects, and everything else PEBL uses to create an experiment. Unlike other languages, a variable need not be declared before it can be used. If you try to access a variable that has not yet been declared, PEBL will return a fatal error that stipulates as such.

3.5.1 Coercion/casting

Variants just hide the representational structure from the user. An actual string resides within the variant that holds a string. A long integer resides within the variant that holds an integer.

PEBL Variants are automatically coerced or cast to the most appropriate inner format. For example, `3232.2 + 33` starts out as a floating point and an integer. The sum is cast to a floating point number. Similarly, `"banana" + 33` starts as a string and an integer, but the combination is a string.

3.5.2 Variable Naming

All variables must begin with a lowercase letter. Any sequence of numbers or letters may follow that letter. If the variable begins with a lowercase 'g', it has global scope; otherwise it has local scope.

3.5.3 Variable Scope

As described above, variables can have either local or global scope. Any variable with global scope is accessible from within any function in your program. A variable with local scope is accessible only from within its own function. Different functions can have local variables with the same name. Generally, it is a good idea to use local variables whenever possible, but using global variables for graphical objects and other complex data types can be intuitive.

3.5.4 Copies and Assignment

Variables may contain various types of data, such as simple types like integers, floating-point ratio numbers, strings; and complex types like lists, windows, sounds, fonts, etc. A variable can be set to a new value, but by design, there are very few ways in which a complex object can be changed once it has been set. For example:

```
woof   <- LoadSound("dog.wav")
meow   <- LoadSound("cat.wav")
dog    <- woof
```

Notice that woof and dog refer to the same sound object. Now you may:

```
PlayBackground(woof)
Wait(50)
Stop(dog)
```

which will stop the sound from playing. If instead you:

```
PlayBackground(woof)
Wait(50)
Stop(meow)
```

woof will play until it is complete or the program ends.

Images provide another example. Suppose you create and add an image to a window:

```
mWindow <- MakeWindow()
mImage  <- MakeImage("test.bmp")
AddObject(mImage, mWindow)
Draw()
```

Now, suppose you create another variable and assign its value to `mImage`:

```
mImage2 <- mImage
Move(mImage2, 200, 300)
Draw()
```

Even though `mImage2` was never added to `mWindow`, `mImage` has moved: different variables now point to the same object. Note that this does not happen for simple (non-object) data types:

```
a <- 33
b <- a
a <- 55
Print(a + "  " + b)
```

This produces the output:

```
55  33
```

This may seem confusing at first, but the consistency pays off in time. The '<-' assignment operator never changes the value of the data attached to a variable, it just changes what the variable points to. PEBL is functional in its handling of simple data types, so you can't, for example, directly modify the contents of a string.

```
a  <- "my string"    #assigns a string literal to a
b  <- a              #makes b refer to a's string literal
a  <- "your string"  #re-assigns a to a new string literal
b  <- a              #makes b refer to a's new string literal
```

There are no 'list surgery' or 'string surgery' functions, like:

```
SetCharacter(a,5,"X")
```

which would theoretically change the string pointed to by 'a' to "yourXstring". If there were, 'b' would also point to "yourXstring".

3.5.5 Passing by Reference and by Value

The discussion in 3.5.4 on copying has implications for passing variables into functions. When a variable is passed into a function, PEBL makes a copy of that variable on which to operate. But, as discussed in 3.5.4, if the variable holds a complex data type (object or a list), the primary data structure allows for direct modification. This is practical: if you pass a window into a function, you do not want to make a copy of that window on which to operate. If the value is a string or a number, a copy of that value is made and passed into the function.

3.6 Functions

The true power of PEBL lies in its extensive library of functions that allow specific experiment-related tasks to be accomplished easily. For the sake of convenience, the library is divided into a number of subordinate libraries. This library structure is transparent to the user, who does not need to know where a function resides in order to use it. Chapter 5 includes a quick reference to functions; Chapter 6 includes a complete alphabetical reference.

3.7 A Simple Program

The previous sections provide everything you need to know to write a simple program. Here is an annotated program:

```
# Any line starting with a # is a comment.  It gets ignored.

------------------------------------------------------
```

```
#Every program needs to define a function called Start() define Start(par)
#Start needs a parameter, just in case braces below contain PEBL statements
{

 ##Assign a number to a variable
 number <- 10

 ##Assign a string to a variable
 hello  <- "Hello World"

 ##Create a global variable (starts with little g)
 gGlobalText <- "Global Text"

 ##Call a user-defined function (defined below).
 value <-  PrintIt(hello, number)
 ##It returned a value

 #Call a built-in function
 Print("Goodbye. " + value)
}

##Define a function with two variables.
define PrintIt(text, number)
{
  #Seed RNG with the current time.
  RandomizeTimer()
  #Generate a random number between 1 and number
  i <- RandomDiscrete(number)  #this is a built-in function

  ##Create a counter variable
  j <- 0
  ##Keep sampling until we get the number we chose.
  while(i != number)
  {
       Print(text + "  " + i + gGlobalText)
       i <- RandomDiscrete(number)
       j <- j + 1
  }

  #return the counter variable.
  return(j)
}

------------------------------------------------------------
```

More sample programs can be found in the `demo/` and `experiments/` directories of the PEBL source tree.

Chapter 4

Overview of Object Subsystems

In PEBL, complex objects are stored and automatically self-managed. These objects include lists, graphical display widgets like images and text displays, fonts, colors, audio files, and input or output files. Objects are created and modified with special functions, but many of their properties available directly for access and modification with a `variable.property` syntax. For example, the position of a textbox is controlled by `.X` and `.Y` properties, and can also be changed with the `Move()` function. To move the label `lab`, which is located at 100,100, to 150,100, you can either do `Move(lab,150,100)` or `lab.X <- 150`. The available properties and accessor function are listed in the decriptions of their relevant objects below.

4.1 Lists

Lists are incredibly useful and flexible storage structures that play an important role in PEBL. A list is simply a series of variables. It is like an array, except that it takes longer to access items later in the list than items at the beginning. But it is much easier to do things like split and combine lists of items than arrays of items. And given the speed of computers, accessing elements of a list is not too costly, unless the list is really long (thousands of items). The `Nth` function can extract items from a list, but it is somewhat costly, and there are often better ways.

For example, suppose you want to print out every item in a list. Looping through, accessing, and printing all the items of a list is a traditional approach:

```
list <- Sequence(1,9,1) #could also be written [1,2,3,4,5,6,7,8,9]
len <- Length(list)
i <- 1
while (i <= len)
```

```
 {
   item <- Nth(list,i)
   Print(item)
   i <- i + 1
}
```

But this is inefficient for many reasons (it could be made more efficient, but only with a loss in clarity). The biggest problem is that the proper element of the list must be found during each iteration, which takes longer as `i` grows. This poses a considerable problem for larger lists.

However, there is an alternative. Items from lists can be iterated over using the '`loop`' command:

```
list <- Sequence(1,9,1)#could also be written [1,2,3,4,5,6,7,8,9]
loop(item, list)
{
  Print(item)
}
```

These two code blocks produce identical output, but in the former block, each item of the list must be found on each iteration, which takes longer as `i` grows. In the latter block, a list item is bound directly to '`item`' on each iteration, so every item on the list takes the same amount of time. Not only is the latter more efficient, it is implemented in fewer lines of code, and so fewer errors (like forgetting to increment `i`) are possible.

A caveat when using lists: Some functions operate on lists to produce new lists (sub-lists, re-ordered lists, etc.). When the lists contain simple data types (numbers, strings, etc.), entirely new data structures are created. But when the data structures are complex (windows, sounds, images, etc.), the objects are not copied. Only new pointers to the original objects are created. So if you change the original object, you may end up accidentally changing the new object. Although that is relatively difficult, because PEBL allows only limited modification of existing data structures, it is still possible. This is a special case of the copy/assignment issue discussed in Section 3.5.4: Copies and Assignment.

4.2 Fonts

PEBL uses truetype fonts for the display of text in labels and other text widgets. In addition to the filename, font objects have the following properties: style (i.e., normal, bold, italic, underline), size (in points), foreground color, background color, and whether it should be rendered anti-aliased.

We distribute a series of high-quality freely available and redistributable fonts, including the Bitstream Vera series, freefont series, and a few others. These include the typeface/files shown below 4.1:

Table 4.1: Typeface/Files Available in PEBL

Filename	Description
	FreeFont Fonts
FreeSans.ttf	Simple Clean sans serif font
FreeSansBold.ttf	
FreeSansOblique.ttf	
FreeSansBoldOblique.ttf	
FreeMono.ttf	Courier-like fontface
FreeMonoBold.ttf	
FreeMonoOblique.ttf	
FreeMonoBoldOblique.ttf	
FreeSerif.ttf	Similar to Times New Roman
FreeSerifBold.ttf	
FreeSerifItalic.ttf	
FreeSerifBoldItalic.ttf	
	Fontforge Fonts
Caliban.ttf	Helvetica-style
CaslonRoman.ttf	Quirky Roman Font series
CaslonBold.ttf	
CaslonItalic.ttf	
Caslon-Black.ttf	
Humanistic.ttf	Sharp, refined fontface
	SIL Fonts
DoulosSILR.ttf	Comprehensive font with roman and cyrillic glyphs
GenR102.ttf	Includes many latin alphabet letters
GenI102.ttf	
CharisSILR.ttf	Like doulos, optimized for printing
CharisSILB.ttf	
CharisSILI.ttf	
CharisSILBI.ttf	
	PEBL Fonts
Stimulasia.ttf	A small set of arrow/boxes
	Bitstream Vera Series
Vera.ttf	Sans serif Roman-style base font
VeraMono.ttf	Sans serif Roman-style mono-spaced base font
VeraSe.ttf	Serif Roman-style base font (similar to times)
VeraBd.ttf	Bold Vera
VeraIt.ttf	Italic Vera
VeraBI.ttf	Bold Italic Vera
VeraMoBd.ttf	Bold Vera Mono
VeraMoIt.ttf	Italic Vera Mono
VeraMoBI.ttf	Bold Italic Vera Mono
VeraSeBd.ttf	Bold Serif Vera

These should always be available for use in experiments. The `fonts.pbl` script in the demo/ directory will display what symbols from each of these fonts looks like.

To use, you need only specify the font name in the `MakeFont()` function:

```
colorRed  <- MakeColor("red")
colorGrey <- MakeColor("grey")
myFont    <- MakeFont("VeraMono.ttf",0,22,colorRed,colorGrey,1)
```

This code makes a red 22-point anti-aliased font on a grey background. Other fonts may be used by specifying their absolute pathname or copying them to the working directory.

Accessible font properties:

```
font.FILENAME
font.BOLD
font.UNDERLINE
font.ITALIC
font.SIZE
font.FGCOLOR
font.BGCOLOR
font.ANTIALIASED
```

4.3 Colors

Colors are PEBL objects. A color can be created by specifying its name using the `MakeColor()` function, or by specifying its RGB values using the `MakeColorRGB()` function. A list of colors and their respective RGB values can be found in the `Colors.txt` file in the documentation directory. There are nearly 800 from which to choose, so you can create just about anything you can imagine.

Accessible color properties:

```
color.RED
color.GREEN
color.BLUE
color.ALPHA
```

4.4 Windows

To run an experiment, you usually need to create a window in which to display stimuli. This is done with the `MakeWindow()` function. `MakeWindow()` will create a grey window by default, or you can specify a color. Currently, an experiment can have only one window.

4.5 Graphical Widgets

Graphical widgets are the building blocks of experimental stimuli. Currently, three widgets are available: images, labels, and textboxes. More complicated widgets are in progress or planned.

To be used, a widget must be created and added to a parent window, and then the parent window must be drawn. You can hide widgets with the `Hide()` function, and show them with the `Show()` function; however, this affects only the visibility of the widget: it is still present and consuming memory. Widgets can be moved around on the parent window using the `Move()` function. `Move()` moves the center of an image or label to the specified pixel, counting from the upper-left corner of the screen. `Move()` moves the upper left-hand corner of textboxes. For the sake of convenience, the `MoveCorner` function is available, which will move an image or label by its upper left-hand corner.

You should remove widgets from their parent window when you are finished using them.

All widgets have several properties available for controlling their behavior.

```
widget.X
widget.Y
widget.WIDTH
widget.HEIGHT
widget.VISIBLE
widget.ROTATION
widget.XZOOM
widget.YZOOM
```

4.6 Images

PEBL can read numerous image types, courtesy of the `SDL_image` library. Use the `MakeImage()` function to read an image into an image object. As images are often used as stimuli, `Move()` centers the image on the specified point. To move by the upper-left hand corner, use the PEBL-defined `MoveCorner()` function:

```
define MoveCorner(object, x, y)
{
 size    <- GetSize(object)
 centerX <- x + First(size)/2
 centerY <- y + Last(size)/2
 Move(object, centerX, centerY)
}
```

Images have all the properties available for widgets, but the width and height can only be read, and not set. Width and height are controlled by the dimensions of the image file.

4.7 Shapes

PEBL allows you to define a number of shape objects that can be added to another widget. A demonstration script exercising these shapes is found in demo/shapes.pbl.

The following is a list of shape and their properties.

4.7.1 Circle

Description: A standard circle. Move commands move the center of the circle to the specified location.

Command: `Circle(<x>,<y>,<r>,<color>,<filled>)`

Properties: .filled = 0,1 (whether it is filled)
.color (color)
.x (x position of center)
.y (y position of center)
.height (read-only height)
.width (read-only width)
.R (radius)

4.7.2 Ellipse

Description: An ellipse, with height and width differing. Cannot be pointed in an arbitrary direction. Move commands move the center of the shape to the specified location.

Command: `Ellipse(<x>,<y>,<rx>,<ry>,<color>,<filled>)`

Properties: .filled = 0,1 (whether it is filled)
.color (color)
.x (x position of center)
.y (y position of center)
.height (read-only height)
.width (read-only width)
.rx (x radius)
.ry (y radius

4.7.3 Square

Description: A square. Move commands move the center of the shape to the specified location.

Command: `Square(<x>,<y>,<size>,<color>,<filled>)`

Properties:
.filled = 0,1 (whether it is filled)
.color (color)
.x (x position of center)
.y (y position of center)
.height (read-only height)
.width (read-only width)
.dx, .dy, .size (Length of side)

4.7.4 Rectangle

Description: A Rectangle. Move commands move the center of the rectangle to the specified location.
Command: `Rectangle(<x>,<y>,<dx>,<dy>,<color>,<filled>)`

Properties: .filled = 0,1 (whether it is filled)
.color (color)
.x (x position of center)
.y (y position of center)
.height (read-only height)
.width (read-only width)
.dx, (width) .dy, (height)

4.7.5 Line

Description: A Line. Move commands move the center of the line to the specified location.
Command: `Line( <x>, <y>,<dx>,<dy>,<color>)`

Properties: .color (color)
.x (x position of start)
.y (y position of start)
.width, (x length)
.height, (y length)

4.7.6 Polygon

Description: An arbitrary polygon.
Command: `Polygon( <x>, <y>,<xpoints>,<ypoints>,<color>,<filled>)`

Properties: .color (color)
.x (x position of start)

.y (y position of start)

4.7.7 Bezier

Description: An arbitrary bezier curve.
Command: `Bezier( <x>, <y>,<xpoints>,<ypoints>,<steps>,<color>)`

Properties: .color (color)
.x (x position of start)
.y (y position of start)

4.8 Text Labels

You can create a text label object with the `MakeLabel()` function, which requires specifying a font, and the foreground and background colors. Labels are only a single line of text. Like images, when you move them, they center on the specified point.

The text inside a label can be extracted with `GetText()` and set with `SetText()`. When you change a text object, it will not appear until the next time you call a `Draw()` function.

Text labels have all the regular widget properties, plus:

```
label.TEXT
label.FONT
```

The `.HEIGHT` and `.WIDTH` accessible, but cannot be changed because they are controlled by the text and the font size.

4.9 Text Boxes

A text box is a graphical widget that contains a body of text. Text automatically wraps when it is too long to fit on a single line. Like labels, the text inside a TextBox can be extracted with `GetText()` and set with `SetText()`. When a text object is changed, it rerenders immediately, but does not appear until the next time a `Draw()` function is called.

Textbox properties:

```
textbox.EDITABLE
textbox.CURSORPOS
```

4.10 User-Editable Text Boxes

Text box editing can be performed using the `GetInput(<textbox>,<escape-key>)` function. This returns the text that is present in the box when the participant

hits the key associated with <escape-key>. <escape-key> is just a text-based code that describes the keypress that should be checked for exit. Typical escape-key options include:

```
"<return>"
"<esc>"
"<backspace>"
"<kp_enter>"
" "
"A"
```

See the Keyboard Entry section below for a more complete list.

Translation from string to keyboard input is still crude, and is handled in src/utility/PEBLUtility.cpp:TranslateString

4.11 Audio

Currently, audio output is very primitive, and there are no facilities for recording or analyzing audio input. Audio .wav files can be loaded with the LoadSound() function, which returns an audio stream object that can be played with either the PlayForeground() or PlayBackground() functions. The PlayForeground() function returns once the sound is finished playing; PlayBackground() returns immediately and the sound plays in a separate thread. When using PlayBackground, playing can be stopped using the Stop() function. If another PlayForeground() or PlayBackground() is then used, the initial sound will immediately terminate and the new file will play. Currently, PEBL can only play one sound at a time.

4.12 Keyboard Entry

PEBL can examine the state of the keyboard, and wait for various keyboard events to happen. Functions such as WaitForKeyDown(), WaitForAnyKeyDown(), etc., allow you to collect responses from subjects. Most keys are specified by their letter name; others have special names:

```
"<left>"
"<up>"
"<down>"
"<right>"
"<enter>"
"<return>"
"<esc>"
"<backspace>" or "<back>"
"<kp_0>" through "<kp_9>", as well as "<kp_period>", "<kp_divide>",
 "<kp_multiply>", "<kp_minus>", "<kp_plus>", "<kp_equals>",
"<kp_enter>" for  keypad keys.
```

`"<insert>","<delete>", "<home>", "<end>","<pageup>","<pagedown>" for other special keys.`

Function keys ¡F1¿ through ¡F15¿.

Also, the traditional "modifier" keys can serve as normal keys:

```
<lshift>, <rshift> <numlock>, <capslock>, <scrollock>,
<rctrl>, <lctrl>, <ralt>,<lalt>,<rmeta>,<lmeta>,<lsuper>,
<rsuper>,<mode>,<compose>
```

4.13 Files

Files are objects that can be read from or written to using several PEBL functions. To use a file object, create one using one of the functions listed below. Each function returns a file object:

```
FileOpenRead()
FileOpenWrite()
FileOpenAppend()
```

For example, you can use the command `myfile <- FileOpenRead("stimuli.txt")` to create '`myfile`', a readable file stream.

Other Functions described below allow filestreams to be written to or read from. When you are finished, you can close a filestream Using the '`FileClose()`' function.

4.14 Network Connections

PEBL has limited ability to open and communicate via TCP/IP connections, either some other system (e.g., for synchronizing with an e.e.g. or eyetracking computer), or another computer running PEBL (e.g., to create multi-subject game theory experiments or to have an experimenter controlling the task from another computer.)

4.14.1 TCP/IP Overview

TCP/IP is a protocol by which computers can talk to one another. It is fairly barebones, and PEBL tries to hide much of its complexity. The information you send from one computer to another is guaranteed to arrive in the correct order, at the potential cost of serious delays, especially if the computers are on different networks or in different locations. Furthermore, connecting PEBL to another computer in this way is a potential security risk. However, the ability to transfer information between computers opens up huge potential for the types of experiments that can be constructed.

4.14.2 Addresses and Ports

To do this, you first must open a network object to communicate with another computer. To do this, you must know (1) the IP number (like 127.0.0.1) or hostname (like myname.myschool.edu) of the computer you want to connect to, and (2) the port you want to connect on. You can even use the protocol to connect to another program running on your own computer, by specifying an IP address of 127.0.0.1, or the hostname "localhost". A port is a number–usually 2 to 5 digits, specifying a type of service on your computer. Many ports are frequently used for specific types of communication, but you can use any port you wish to communicate, as long as both computers know this port. Most ports on your computer should be blocked by default, so you may need to turn off your firewall or allow your chosen port to pass through the security or you may have trouble communicating.

To allow two PEBL programs to communicate, you need to decide that one computer is the "server" and the other is the "client". On the server, you execute the function `WaitForNetworkConnection(port)`, which listens on the specified port until the client tries to connect. After the server is started, the client calls `ConnectToHost(hostname, port)` or `ConnectToIP(ipnum, port)`, depending upon whether you are using the hostname or ip address. Typically, ip numbers are specified by four three-digit numbers separated by dots, like 196.168.0.1. This actually represents a 4-byte integer, and this 4-byte integer is what `ConnectToIP()` expects. To create that integer, use the function `ConvertIPString(ipnum)`, which accepts an IP address specified in a string. So, you can use:

```
net <- ConnectToIP(ConvertIPString( "127.0.0.1"), 1234)
```

to create a connection to another program listening on port 1234 on your own computer. These functions all return a network object (e.g., `net`) that must be used in later communication.

4.14.3 Sending and Receiving Data

Once connected, the distinction between client and server essentially disappears. However, to communicate, one computer must send data with the `SendData(net, data)`, and the other must receive the data, using the `GetData(net, size)` function. PEBL can only send text strings, and you must know the length of the message you want to receive. More complex communication can be done by creating a set of PEBL functions that encapsulate messages into text strings with templated headers that specify the message length. Then, to receive a message, you first read the fixed-length header, determine how much more data needs to be read, then read in the rest of the data.

4.14.4 Closing networks

If you are using a network connection to synchronize timing of two computers, you probably want to close the network connection with `CloseNetworkConnection(net)`

after you have synchronized, to avoid any extra overhead.

A simple example of an experiment that uses TCP/IP to communicate is the NIM game in demo/nim.pbl.

4.15 The Event Loop

To assist in testing for multiple input events simultaneously, PEBL implements an event loop that will quickly scan multiple conditions and execute proper results whenever any one condition is met. This is currently primarily a back-end system which will be developed more in the future.

4.16 Errors and Warnings

PEBL does a great deal of error-checking to ensure that your program will run. If you crash with a segmentation fault, this is an error and you should report it. When a fatal error or non-fatal warning occurs, PEBL attempts to identify the location in your input file that led to the warning. On Linux, the warning and this location are printed to the command-line upon exit; on MS Windows, they are printed to the file `stderr.txt`.

You can use the error system in your own scripts with the `SignalFatalError()` function. This is especially useful in combination with the functions testing the type of object passed into the function. To ensure proper processing and ease of debugging, test the format of an argument passed into a function:

```
define MyFunction(par)
 {
   if(not IsList(par))
   {SignalFatalError("MyFunction was passed a non-list variable.")}

   ## Do stuff here.
 }
```

4.17 Paths and Path Searching

Numerous functions and objects open files on your computer to read in information such as graphics, sounds, fonts, program files, and text files. When you attempt to open a file, PEBL will search in a number of places, in this order:

- The (current) working directory
- The directory of each file specified in the command line arguments
- media/fonts
- media/sounds

- media/images
- media/text

You can also specify other paths to be searched by specifying them on the command line. Be sure to end the directory with whatever is appropriate for your platform, e.g. '"' on Microsoft Windows or '/' on Linux.

4.18 Provided Media Files

PEBL comes with various media files that can be specified from any script without including the complete path. If a user's file has the same name, it will be loaded before the PEBL-provided version. Table 4.2 describes the files included.

Table 4.2: Media Files Provided with PEBL

Name	Description
In 'media/fonts/':	
Listing of fonts appears in Table 4.1	
In 'media/images/':	
`pebl.bmp`	Demonstration bitmap image
`pebl.png`	Demonstration PNG image
`smiley-small.png`	25x25 smiley face
`frowney-small.png`	25x25 frowney face
`smiley-large.png`	100x100 smiley face
`frowney-large.png`	100x100 frowney face
In 'media/sounds/':	
`buzz500ms.wav`	A 500-ms buzzer
`chirp1.wav`	A chirp stimulus
`boo.wav`	A really bad booing sound
`cheer.wav`	A pretty lame cheering sound
In 'media/text/':	
`Consonants.txt`	List of all consonants, both cases
`Digits.txt`	List of digits 0-9
`DigitNames.txt`	List of digit names
`Letters.txt`	All letters, both cases
`Lowercase.txt`	Lowercase letters

Name	Description
`LowercaseConsonants.txt`	Lowercase Consonants
`LowercaseVowels.txt`	Lowercase Vowels
`Uppercase.txt`	Uppercase Letters
`UppercaseConsonants.txt`	Uppercase Consonants
`UppercaseVowels.txt`	Uppercase Vowels
`Vowels.txt`	Vowels (both cases)

Additionally, the PEBL Project distributes a number of other media files separately from the base system. These are available for separate download on the pebl website (http://pebl.sourceforge.net), and include a set of images (including shapes and sorting-task cards), and a set of auditory recordings (including beeps, the digits 0-10, and a few other things).

4.19 Special Variables

There are a number of special variables that be set by PEBL, and can later be accessed by an experiment. These are described in table 4.3.

Table 4.3: Special Variables in PEBL

Name	Purpose
`gKeepLooping`	Controls continued execution in event loop. (Not currently useful).
`gSleepEasy`	Sets 'busy-waiting' to be either on or off. Busy-waiting can improve timing, but is often not needed and pegs CPU.
`gVideoWidth`	The width in pixels of the display (set by default or command-line option). Changing this before calling `MakeWindow` will change display width, if that width is available.
`gVideoHeight`	The height in pixels of the display (set by default or command-line). Changing this before calling `MakeWindow()` will change the display height, if that height is available.
`gVideoDepth`	The bit depth of the video.
`gSubNum`	A global variable set to whatever follows the `--s` or `--S` command-line argument. If no argument is given, defaults to 0.

Chapter 5

Function Quick Reference

Table 5.1 lists the functions available for use with PEBL. Those that are unimplemented are noted as such. If you want the functionality of an unimplemented function, or want functionality not provided in any of these functions, contact us, or better yet, contribute to the PEBL project by implementing the function yourself.

Table 5.1: Function Quick Reference

Name	Arguments	Description
	Math Functions	
Log10	<num>	Log base 10 of <num>
Log2	<num>	Log base 2 of <num>
Ln	<num>	Natural log of <num>
LogN	<num> <base>	Log base <base> of <num>
Exp	<pow>	e to the power of <pow>
Pow	<num> <pow>	<num> to the power of <pow>
Sqrt	<num>	Square root of <num>
NthRoot	<num> <root>	<num> to the power of 1/<root>
Tan	<deg>	Tangent of <deg> degrees
Sin	<deg>	Sine of <deg> degrees
Cos	<deg>	Cosine of <deg> degrees
ATan	<num>	Inverse Tan of <num>, in degrees
ASin	<num>	Inverse Sine of <num>, in degrees
ACos	<num>	Inverse Cosine of <num>, in degrees
DegToRad	<deg>	Converts degrees to radians
RadToDeg	<rad>	Converts radians to degrees
Round	<num> <sig>	Rounds <num> to <sig> significant digits
Floor	<num>	Rounds <num> down to the next integer

Name	Arguments	Description
Ceiling	<num>	Rounds <num> up to the next integer
AbsFloor	<num>	Rounds <num> toward 0 to an integer
Mod	<num> <mod>	Returns <num> mod <mod> or remainder of <num>/<mod>
Div	<num> <mod>	Returns round(<num>/<mod>)
ToInteger	<num>	Rounds a number to an integer, and changes internal representation
ToFloat	<num>	Converts number to internal floating-point representation
ToNumber	<>	
ToString	<num>	Converts a numerical value to a string representation
Sign	<num>	Returns +1 or -1, depending on sign of argument
Abs	<num>	Returns the absolute value of the number
CumNormInv	<p>	Returns accurate numerical approximation of cumulative normal inverse.
NormalDensity	<x>	Returns density of standard normal distribution.
SDTDPrime	<hr>,<far>	Computes SDT dprime.
SDTBeta	<hr>,<far>	Computes SDT beta.
Order	<list>	Returns a list of integers representing the order of <list>
Rank	<list>	Returns integers representing the ranked indices of the numbers of<list>
Median	<list>	Returns the median value of the numbers in <list>
Min	<list>	Returns the smallest of <list>
Max	<list>	Returns the largest of <list>
StDev	<list>	Returns the standard dev of <list>
Sum	<list>	Returns the sum of the numbers in <list>
Median	<list>	Returns the median of a set of values
Quantile	<list> <num>	Returns the <num> quantile of the numbers in <list>
SummaryStats	<data>,<cond>	Returns statistics (cond,N,median,mean,sd) computed on data for each distinct value of <cond>
SeedRNG	<num>	Seeds the random number generator with <num> to reproduce a random sequence
RandomizeTimer	-	Seeds the RNG with the current time

Name	Arguments	Description
Random	-	Returns a random number between 0 and 1
RandomDiscrete	<num>	Returns a random integer between 1 and <num>
RandomUniform	<num>	Returns a random floating-point number between 0 and <num>
RandomNormal	<mean> <stdev>	Returns a random number according to the standard normal distribution with <mean> and <stdev>
RandomExponential	<mean>	Returns a random number according to exponential distribution with mean <mean> (or decay 1/mean)
RandomLogistic	<p>	Returns a random number according to the logistic distribution with parameter <p>
RandomLogNormal	<median> <spread>	Returns a random number according to the log-normal distribution with parameters <median> and <spread>
RandomBinomial	<p> <n>	Returns a random number according to the Binomial distribution with probability <p> and repetitions <n>
RandomBernoulli	<p>	Returns 0 with probability (1-<p>) and 1 with probability <p>
ZoomPoints	<[xs,yy]>, <xzoom>, <yzoom>	Zooms a set of points in 2 directions
ReflectPoints	<[xs,yy]>	Reflects points on vertical axis
RotatePoints	<[xs,yy]>,<angle>	Rotates point <angle> degrees
File/NetworkStream Functions		
Print	<value>	Prints <value> to stdout, appending a new line afterwards. stdout is the console (in Linux) or the file stdout.txt (in Windows)
Print_	<value>	Prints <value> to stdout, without appending a newline afterwards
PrintList	<value>	Prints <list>, getting rid of '[', ']' and ',' characters.
Format	<object> <size>	Prints a number in size spaces by truncating or padding
FileOpenRead	<filename>	Opens a filename, returning a stream to be used for reading information

Name	Arguments	Description
`FileOpenWrite`	`<filename>`	Opens a filename, returning a stream that can be used for writing information. Overwrites if file already exists
`FileOpenAppend`	`<filename>`	Opens a filename, returning a stream that can be used for writing info. Appends if the file already exists, opens if file does not
`FileClose`	`<filestream>`	Closes a filestream variable. Pass the variable name, not the filename
`FilePrint`	`<filestream>` `<value>`	Like `Print`, but to a file.
`FilePrint_`	`<filestream>` `<value>`	Like `Print_`, but to a file.
`FilePrintList`	`<file><list>`	Prints `<list>` to `<file>`, getting rid of '[', ']' and ',' characters.
`FileReadCharacter`	`<filestream>`	Reads and returns a single character from a filestream
`FileReadWord`	`<filestream>`	Reads and returns a 'word' from a file; the next connected stream of characters not including a `' '` or a newline. Will not read newline characters
`FileReadLine`	`<filestream>`	Reads and returns a line from a file; all characters up until the next newline or the end of the file
`FileReadList`	`<filename>`	Given a filename, will open it, read in all the items into a list (one item per line), and close the file afterwards
`FileReadTable`	`<filename>` `<opt-sep>`	Like `FileReadList`, but reads in tables. Optionally, specify a token separator
`FileReadText`	`<filename>`	Reads all of the text in the file into a variable
`EndOfLine`	`<filestream>`	Returns true if at end of line
`EndOfFile`	`<filestream>`	Returns true if at the end of a file
`ConnectToIP`	`<ip> <port>`	Connects to a port on another computer, returning network object.
`ConnectToHost`	`<hostname>` `<port>`	Connects to a port on another computer, returning network object.
`WaitForNetworkConnection`	`<port>`	Listens on a port until another computer connects, returning a network object
`CloseNetworkConnection`	`<network>`	Closes network connection

Name	Arguments	Description
SendData	<network> <datastring>	Sends a data string over connection.
GetData	<network> <length>	return a string from network connection
ConvertIPString	<ip-as-string>	Converts an ip-number-as-string to usable address
Graphical Objects Functions		
MakeWindow	<colorname>	Creates main window, in color named by argument, or grey if no argument is named
MakeImage	<filename>	Creates an image by reading in an image file (jpg, gif, png, bmp, etc.)
MakeLabel	<text> <font>	Creates a single line of text filled with <text> written in font <font>
MakeTextBox	<text> <font> <width> <height>	Creates a sized box filled with <text> written in font <font>
EasyLabel	<text> <x><y> <win><fontsize>	Creates a single line of text and adds it to win at <x><y>
EasyTextBox	<text> <x> <y> <win> <fontsize> <width> <height>	Creates a textbox and adds it to <win> at <x><y>
MakeColor	<colorname>	Creates a color based on a color name
MakeColorRGB	<red> <green> <blue>	Creates a color based on red, green, and blue values
MakeFont	<ttf_filename> <style> <size> <fgcolor> <bgcolor> <anti-aliased>	Creates a font which can be used to make labels
SetCursorPosition	<textbox> <position>	Move the editing cursor in a textbox
GetCursorPosition	<textbox>	Gets the position of the editing cursor
SetEditable	<textbox> <status>	Turns on or off the editing cursor
GetText	<textobject>	Returns the text in a textbox or label
GetInput	<textbox> <escape-key>	Allows a textbox to be edited by user, returning its text when <escape-key> is pressed.
SetText	<textobject>, <text>	Sets the text in a textbox or label

Name	Arguments	Description
SetFont	<textobject>, <font>	Changes the font of a text object
Move	<object> <x> <y>	Move an object (e.g., an image or a label to an x,y location)
MoveCorner	<object> <x> <y>	Moves an image or label by its upper corner.
GetSize	<object>	Returns a list of dimensions <x,y> of a graphical object.
AddObject	<object> <parent>	Adds an object to a parent object (window)
RemoveObject	<object> <parent>	Removes an object from a parent window
Show	<object>	Shows an object
Hide	<object>	Hides an object
ShowCursor	<object>	Hides or show mouse cursor.
GetMouseCursorPosition		Gets [x,y] position of mouse
SetMouseCursorPosition	<x>,<y>	Sets x,y position of mouse
ShowCursor	<object>	Hides or show mouse cursor.
Draw	<object>	Redraws a widget and its children
DrawFor	<object> <cycles>	Draws for exactly <cycles> cycles, then returns
Circle	<x> <y> <r> <color> <filled>	Creates circle with radius r centered at position x,y
Ellipse	<x> <y> <rx> <ry><color> <filled>	Creates ellipse with radii rx and ry centered at position x,y
Square	<x> <y> <size> <color> <filled>	Creates square with width size centered at position x,y
Rectangle	<x> <y> <dx> <dy><color> <filled>	Creates rectangle with size (dx, dy) centered at position x,y
Line	<x> <y> <dx> <dy> <color>	Creates line starting at x,y and ending at x+dx, y+dy
PrintProperties	<object>	Prints a list of all available properties of an object (for debugging)
Polygon	<x> <y> <xpoints> <ypounts> <color><filled>	Creates polygon centered at x,y with relative points <xpoints>,<ypoints>
Bezier	<x> <y> <xpoints> <ypounts> <steps> <color>	Creates bezier curve centered at x,y with relative points <xpoints>,<ypoints>

Name	Arguments	Description
BlockE	<x> <y> <h> <w> <thickness> <orientation> <color>	Creates a block E as a useable polygon which can be added to a window directly.
Plus	<x> <y> <size> <w> <color>	Creates a plus sign as a useable polygon which can be added to a window directly.
MakeStarPoints	<r_outer> <r_inner> <npeaks>	Creates points for a star, which can then be fed to Polygon
MakeNGonPoints	<radius> <npeaks>	Creates points for a polygon, which can then be fed to Polygon
Sound Objects Functions		
LoadSound	<filename>	Loads a soundfile from the filename, returning a variable that can be played
PlayForeground	<sound>	Plays the sound 'in the foreground', not returning until the sound is complete
PlayBackground	<sound>	Plays the sound 'in the background', returning immediately
Stop	<sound>	Stops a sound playing in the background from playing
Misc Event Functions		
GetTime	<>	Gets a number, in milliseconds, representing the time since the PEBL program began running.
Wait	<time>	Pauses execution for <time> ms
IsKeyDown	<keyval>	Determines whether the key associated with <keyval> is down
IsKeyUp	<keyval>	Determines whether the key associated with <keyval> is up
IsAnyKeyDown	<>	Determines whether any key is down.
WaitForKeyDown	<keyval>	Waits until <keyval> is detected to be in the down state
WaitForAnyKeyDown	<>	Waits until any key is detected in down state
WaitForKeyUp	<keyval>	Waits until <keyval> is in up state.
WaitForAllKeysUp		Waits until all keys are in up state
WaitForAnyKeyDownWithTimeout	<time>	Waits for a key to be pressed, but only for <time> ms
WaitForKeyListDown		

Name	Arguments	Description
	<list-of-keyvals>	Waits until one of the keys is in down state
WaitForKeyPress	<key>	Waits until <key> is pressed
WaitForAnyKeyPress	<>	Waits until any key is pressed
WaitForKeyRelease	<key>	Waits until <key> is released
WaitForListKeyPress	<list-of-keys>	Waits until one of <list-of-keys> is pressed
WaitForListKeyPressWithTimeout	<list-of-keyvals> <timeout> <type>	Waits for either a key to be pressed or a time to pass.
WaitForMouseButton		Waits until any of the mouse buttons is pressed or released, and returns message indicating what happened
RegisterEvent	<>	NOT IMPLEMENTED
StartEventLoop	<>	NOT IMPLEMENTED
ClearEventLoop	<>	NOT IMPLEMENTED
SignalFatalError	<message>	Halts execution, printing out message
TranslateKeyCode	<>	Converts a keycode to a key name
TimeStamp		Returns a string containing the current date and time
GetPEBLVersion	<>	Returns a string indicating which version of PEBL you are using
GetNIMHDemographics	<code> <window> <file>	Asks NIMH-related questions
GetSubNum	<window>	Asks user to enter subject number
IsNumber	<variant>	Tests whether <variant> is a number
IsInteger	<variant>	Tests whether <variant> is an integer-type number
IsFloat	<variant>	Tests whether <variant> is a floating-point number
IsString	<variant>	Tests whether <variant> is a string
IsList	<variant>	Tests whether <variant> is a List
IsTextBox	<variant>	Tests whether <variant> is a TextBox
IsImage	<variant>	Tests whether <variant> is an Image
IsLabel	<variant>	Tests whether <variant> is a Text Label
IsAudioOut	<variant>	Tests whether <variant> is a AudioOut stream
IsFont	<variant>	Tests whether <variant> is a Font
IsColor	<variant>	Tests whether <variant> is a Color
IsFileStream	<variant>	Tests whether <variant> is a FileStream
IsWidget	<variant>	Tests whether <variant> is any Widget

Name	Arguments	Description
IsWindow	<variant>	Tests whether <variant> is any Window
IsShape	<variant>	Tests whether <variant> is any drawing shape, such as a circle, square or polygon
List Manipulation Functions		
Shuffle	<list>	Returns a new list with the items in list shuffled randomly.
ShuffleRepeat	<list> <times>	Generates a list of n shuffled versions of <list>
ShuffleWithoutAdjacents	<nested-list>	Shuffle specifying items that should not appear adjacently
Repeat	<item> <n>	Repeats an item n times in a list
RepeatList	<list> <n>	Makes a new list containing the elements of <list> repeated <n> times
Sequence	<start> <end> <step>	Makes a sequence of numbers from <start> to <end>, with <step>-sized increments
ChooseN	<list> <n>	Returns a sublist of <n> items from a list, in the order they appear in the original list
SampleN	<list> <n>	Returns a randomly-ordered sublist of <n> items from a list
SampleNWithReplacement	<list> <n>	Returns a sublist of <n> items from a list
DesignLatinSquare	<list1> <list2>	
LatinSquare	<list>	A simple latin square constructor
DesignGrecoLatinSquare	<list1> <list2> <list3>	
DesignBalancedSampling	<list> <number>	
DesignFullCounterbalance	<list1> <list2>	
CrossFactorWithoutDuplicates	<list>	Returns a list of all pairs of items in the list, excluding pairs that where an element appears twice.
Rotate	<list> <n>	Rotates a list by <n> items.
FoldList	<list> <n>	Folds list into length-n sublists.
Flatten	<list>	Flattens a nested list completely
FlattenN	<list> <n>	Flattens n levels of a nested list
Length	<list>	Returns the number of elements in a list.

Name	Arguments	Description
First	<list>	Returns the first item in a list.
Last	<list>	Returns the last item in a list.
Merge	<list1> <list2>	Combines two lists.
Append	<list> <item>	Adds <item> to <list>
List	<item1> <item2>...	Makes a list out of items
Sort	<list>	Sorts a list by its values.
SortBy	<list> <key>	Sorts list by the values in <key>
Nth	<list> <n>	Returns the nth item in a list.
Subset	<list> <list-of-indices>	
ExtractListItems	<list> <list-of-indices>	
IsMember	<item> <list>	Checks whether <item> is a member of <list>
Replace	<template> <replacementList>	Replaces items in a data structure
Lookup	<key> <keylist> <database>	returns element in <database> corresponding to element of <keylist> that matches <key>.
RemoveDuplicates		NOT IMPLEMENTED
MakeMap		NOT IMPLEMENTED
Transpose	<list-of-lists>	Transposes a list of equal-length lists.
SubList	<list> <start> <finish>	Returns a sublist of a list.
Remove	<list> <n>	Removes an item from a list. Unimplemented
ListToString	<list>	Concatenates all elements of a list into a single string
String Management Functions		
CR	<num>	Returns string with <num> linefeeds.
Tab	<num>	Returns string with <num> tabs.
Format	<value> <num>	Makes string from value exactly <num> characters by truncating or padding.
Uppercase	<string>	Returns uppercased string
Lowercase	<string>	Returns lowercased string
ReplaceChar	<string> <char> <char2>	Substitutes <char2> for <char> in <string>.
SplitString	<string> <split>	Splits <string> into a list of <split>-delimited substrings

Name	Arguments	Description
StringLength	<string>	Returns the length of a string
SubString	<string> <position> <length>	Returns a substring
FindInString	<string> <key>	Returns position of <key> in <string>

Chapter 6

Detailed Function and Keyword Reference.

6.1 Symbols

Name/Symbol: `+`

Description: Adds two expressions together. Also, concatenates strings together.

Usage:
```
<num1> + <num2>
<string1> + <string2>
<string1> + <num1>
```
Using other types of variables will cause errors.

Example:
```
33 + 322                    --> 355
"Hello" + " " + "World"     --> "Hello World"
"Hello" + 33 + 322.5        --> "Hello355.5"
33 + 322.5 + "Hello"        --> "33322.5Hello"
```

See Also: `-`, `ToString()`

Name/Symbol: `-`

Description: Subtracts one expression from another

Usage: `<num1> - <num2>`

Example:

See Also:

Name/Symbol: `/`

Description: Divides one expression by another

Usage: `<expression> / <expression>`

Example:
```
333 / 10    # == 33.3
```

See Also:

Name/Symbol: `*`

Description: Multiplies two expressions together

Usage: `<expression> * <expression>`

Example:
```
32 * 2 # == 64
```

See Also:

Name/Symbol: `^`

Description: Raises one expression to the power of another expression

Usage: `<expression> ^ <expression>`

Example:
```
25 ^ 2  # == 625
```

See Also: `Exp`, `NthRoot`

Name/Symbol: `;`

Description: Finishes a statement, can start new statement on the same line (not needed at end of line)

Usage:

Example:

See Also:

Name/Symbol: `#`

Description: Comment indicator; anything until the next CR following this character is ignored

Usage:

Example:

See Also:

Name/Symbol: `<-`

Description: The assignment operator. Assigns a value to a variable
N.B.: This two-character sequence takes the place of the '`=`' operator found in many programming languages.

Usage:

Example:

See Also:

Name/Symbol: `( )`

Description: Groups mathematical operations

Usage: `(expression)`

Example: `(3 + 22) * 4  # == 100`

See Also:

Name/Symbol: `{ }`

Description: Groups a series of statements

Usage:
```
{ statement1
  statement2
  statement3
}
```

Example:

See Also:

Name/Symbol: `[ ]`

Description: Creates a list. Closing `]` must be on same line as last element of list, even for nested lists.

Usage:
```
[<item1>, <item2>, ....]
```

Example:
```
[]                    #Creates an empty list
[1,2,3]               #Simple list
[[3,3,3],[2,2],0]     #creates a nested list structure
```

See Also: `List()`

Name/Symbol: `<`

Description: Less than. Used to compare two numeric quantities.

Usage:
```
3 < 5
3 < value
```

Example:
```
if(j < 33)
{
  Print ("j is less than 33.")
}
```

See Also: `>`, `>=`, `<=`, `==`, `~=`, `!=`, `<>`

Name/Symbol: `>`

Description: Greater than. Used to compare two numeric quantities.

Usage:
```
5 > 3
5 > value
```

Example:
```
if(j > 55)
{
 Print ("j is greater than 55.")
}
```

See Also: `<`, `>=`, `<=`, `==`, `~=`, `!=`, `<>`

Name/Symbol: `<=`

Description: Less than or equal to.

Usage:
```
3<=5
3<=value
```

Example:
```
if(j <= 33)
{
 Print ("j is less than or equal to 33.")
}
```

See Also: `<, >, >=, ==, ~=, !=, <>`

Name/Symbol: `>=`

Description: Greater than or equal to.

Usage:
```
5>=3
5>=value
```

Example:
```
if(j >= 55)
{
 Print ("j is greater than or equal to 55.")
}
```

See Also: `<, >, <=, ==, ~=, !=, <>`

Name/Symbol: `==`

Description: Equal to.

Usage: `4 == 4`

Example: `2 + 2 == 4`

See Also: `<, >, >=, <=, ~=, !=, <>`

Name/Symbol: `<>, !=, ~=`

Description: Not equal to.

Usage:

Example:

See Also: `<, >, >=, <=, ==`

6.2 A

Name/Symbol: `Abs()`

Description: Returns the absolute value of the number.

Usage: `Abs(<num>)`

Example:
```
Abs(-300)   # ==300
Abs(23)     # ==23
```

See Also: `Round()`, `Floor()`, `AbsFloor()`, `Sign()`, `Ceiling()`

Name/Symbol: `AbsFloor()`

Description: Rounds `<num>` toward 0 to an integer.

Usage: `AbsFloor(<num>)`

Example:
```
AbsFloor(-332.7)    # == -332
AbsFloor(32.88)     # == 32
```

See Also: `Round()`, `Floor()`, `Abs()`, `Sign()`, `Ceiling()`

Name/Symbol: `ACos()`

Description: Inverse cosine of `<num>`, in degrees.

Usage: `ACos(<num>)`

Example:

See Also: `Cos()`, `Sin()`, `Tan()`, `ATan()`, `ATan()`

Name/Symbol: `AddObject()`

Description: Adds a widget to a parent window.

Usage:

Example:

See Also: `RemoveObject()`

Name/Symbol: `and`

Description: Logical and operator.

Usage: `<expression> and <expression>`

Example:

See Also: `or`, `not`

Name/Symbol: `Append`

Description: Appends an item to a list. Useful for constructing lists in conjunction with the loop statement.

Usage: `Append(<list>, <item>)`

Example:
```
list <- Sequence(1,5,1)
double  <- []
loop(i, list)
{
 double <- Append(double, [i,i])
}
Print(double)
# Produces [[1,1],[2,2],[3,3],[4,4],[5,5]]
```

See Also: `List()`, `[ ]`, `Merge()`

Name/Symbol: `ASin()`

Description: Inverse Sine of `<num>`, in degrees.

Usage: `ASin(<num>)`

Example:

See Also: `Cos()`, `Sin()`, `Tan()`, `ATan()`, `ACos()`, `ATan()`

Name/Symbol: `ATan`

Description: Inverse Tan of `<num>`, in degrees.

Usage:

Example:

See Also: `Cos()`, `Sin()`, `Tan()`, `ATan()`, `ACos()`, `ATan()`

6.3 B

Name/Symbol: `Bezier`

Description: Creates a smoothed line through the points specified by `<xpoints>`, `<ypoints>`. The lists `<xpoints>` and `<ypoints>` are adjusted by `<x>` and `<y>`, so they should be relative to 0, not the location you want the points to be at.

Like other drawn objects, the bezier must then be added to the window to appear. ¡steps¿ denotes how smooth the approximation will be.

Usage:
```
Bezier(<x>,<y>,<xpoints>,<ypoints>,<steps>,<color>)
```

Example:
```
win <- MakeWindow()
 #This makes a T
 xpoints <- [-10,10,10,20,20,-20,-20,-10]
 ypoints <- [-20,-20,40,40,50,50,40,40]
p1 <-    Bezier(100,100,xpoints, ypoints,5, MakeColor("black"))
AddObject(p1,win)
Draw()
```

See Also: `BlockE()`, `Polygon()`, `MakeStarPoints()`, `MakeNGonPoints()`

Name/Symbol: `BlockE`

Description: Creates a polygon in the shape of a block E, pointing in one of four directions. Arguments include position in window.

- `<x>` and `<y>` is the position of the center
- `<h>` and `<w>` or the size of the E in pixels
- `<thickness>` thickness of the E
- `<direction>` specifies which way the E points: 1=right, 2=down, 3=left, 4=up.
- `<color>` is a color object (not just the name)

Like other drawn objects, the Block E must then be added to the window to appear.

Usage:
```
BlockE(x,y,h,w,thickness,direction,color)
```

Example:
```
win <- MakeWindow()
e1 <- BlockE(100,100,40,80,10,1,MakeColor("black"))
AddObject(e1,win)
Draw()
```

See Also: `Plus()`, `Polygon()`, `MakeStarPoints()`, `MakeNGonPoints()`

Name/Symbol: `break`

Description: Breaks out of a loop immediately.

Usage: break

Example:
```
loop(i ,[1,3,5,9,2,7])
{
 Print(i)
 if(i == 3)
        {
         break
        }
}
```

See Also: `loop`, `return`

6.4 C

Name/Symbol: `Ceiling()`

Description: Rounds `<num>` up to the next integer.

Usage: `Ceiling(<num>)`

Example:
```
Ceiling(33.23)   # == 34
Ceiling(-33.02)  # == -33
```

See Also: `Round()`, `Floor()`, `AbsFloor()`, `Ceiling()`

Name/Symbol: `ChooseN()`

Description: Samples `<number>` items from list, returning a list in the original order. Items are sampled without replacement, so once an item is chosen it will not be chosen again. If `<number>` is larger than the length of the list, the entire list is returned in order. It differs from `SampleN` in that `ChooseN` returns items in the order they appeared in the originial list, but `SampleN` is shuffled.

Usage: `ChooseN(<list>, <n>)`

Example:
```
ChooseN([1,1,1,2,2], 5)     # Returns 5 numbers
ChooseN([1,2,3,4,5,6,7], 3) # Returns 3 numbers from 1 and 7
```

See Also: `SampleN()`, `SampleNWithReplacement()`, `Subset()`

Name/Symbol: `Circle()`

Description: Creates a circle for graphing at x,y with radius r. Circles must be added to a parent widget before it can be drawn; it may be added to widgets other than a base window. The properties of circles may be changed by accessing their properties directly, including the FILLED property which makes the object an outline versus a filled shape.

Usage: `Circle(<x>, <y>, <r>,<color>)`

Example:
```
c <- Circle(30,30,20, MakeColor(green))
AddObject(c, win)
Draw()
```

See Also: `Square()`, `Ellipse()`, `Rectangle()`, `Line()`

Name/Symbol: `CloseNetworkConnection()`

Description: Closes network connection

Usage: `CloseNetwork(<network>)`

Example:
```
net <- WaitForNetworkConnection("localhost",1234)
SendData(net,"Watson, come here. I need you.")
CloseNetworkConnection(net)
```
Also see nim.pbl for example of two-way network connection.

See Also: `ConnectToIP`, `ConnectToHost`, `GetData`, `WaitForNetworkConnection`, `SendData`, `ConvertIPString`

Name/Symbol: `ConnectToHost()`

Description: Connects to a host computer waiting for a connection on ¡port¿, returning a network object that can be used to communicate. Host is a text hostname, like `"myname.indiana.edu"`, or use `"localhost"` to specify your current computer.

Usage: `ConnectToHost(<hostname>,<port>)`

Example: See nim.pbl for example of two-way network connection.

```
net <- ConnectToHost("localhost",1234)
dat <- GetData(net,20)
Print(dat)
CloseNetworkConnection(net)
```

See Also: `ConnectToIP`, `GetData`, `WaitForNetworkConnection`, `SendData`, `ConvertIPString`, `CloseNetworkConnection`

Name/Symbol: `ConnectToIP()`

Description: Connects to a host computer waiting for a connection on `<port>`, returning a network object that can be used to communicate. `<ip>` is a numeric ip address, which must be created with the `ConvertIPString(ip)` function.

Usage: `ConnectToIP(<ip>,<port>)`

Example: See nim.pbl for example of two-way network connection.

```
ip <- ConvertIPString("192.168.0.1")
net <- ConnectToHost(ip,1234)
dat <- GetData(net,20)
Print(dat)
CloseNetworkConnection(net)
```

See Also: `ConnectToHost`, `GetData`, `WaitForNetworkConnection`, `SendData`, `ConvertIPString`, `CloseNetworkConnection`

Name/Symbol: `ConvertIPString()`

Description: Converts an IP address specified as a string into an integer that can be used by ConnectToIP.

Usage: `ConvertIPString(<ip-as-string>)`

Example: See nim.pbl for example of two-way network connection.

```
ip <- ConvertIPString("192.168.0.1")
net <- ConnectToHost(ip,1234)
dat <- GetData(net,20)
Print(dat)
CloseNetworkConnection(net)
```

See Also: `ConnectToHost`, `ConnectToIP`, `GetData`, `WaitForNetworkConnection`, `SendData`, `ConvertIPString`, `CloseNetworkConnection`

Name/Symbol: `Cos()`

Description: Cosine of `<deg>` degrees.

Usage:

Example:
```
Cos(33.5)
Cos(-32)
```

See Also: `Sin()`, `Tan()`, `ATan()`, `ACos()`, `ATan()`

Name/Symbol: `CR()`

Description: Produces `<number>` linefeeds which can be added to a string and printed or saved to a file.

Usage: `CR(<number>)`

Example:
```
          Print("Number: "  Tab(1) + number  + CR(2))
          Print("We needed space before this line.")
```

See Also: `Format()`, `Tab()`

Name/Symbol: `CrossFactorWithoutDuplicates()`

Description: This function takes a single list, and returns a list of all pairs, excluding the pairs that have two of the same item. To achieve the same effect but include the duplicates, use: `DesignFullCounterBalance(x,x)`.

Usage: `CrossFactorWithoutDuplicates(<list>)`

Example:
```
CrossFactorWithoutDuplicates([a,b,c])
# == [[a,b],[a,c],[b,a],[b,c],[c,a],[c,b]]
```

See Also: `DesignFullCounterBalance()`, `Repeat()`, `DesignBalancedSampling()`, `DesignGrecoLatinSquare()`, `DesignLatinSquare()`, `RepeatList()`, `LatinSquare() Shuffle()`

Name/Symbol: `CumNormInv()`

Description: This function takes a probability and returns the corresponding z-score for the cumulative standard normal distribution. It uses an accurate numerical approximation from:
`http://home.online.no/ pjacklam/notes/invnorm`

Usage: `CumNormInv(<p>)`

Example:

```
 Print(CumNormInv(0))    #= NA
  Print(CumNormInv(.01)) #= -2.32634
  Print(CumNormInv(.5))  #= 0
  Print(CumNormInv(.9))  #= 1.28
  Print(CumNormInv(1))   #= NA
```

See Also: `NormalDensity()`, `RandomNormal()`

6.5 D

Name/Symbol: `define`

Description: Defines a user-specified function.

Usage:

```
define functionname (parameters)
{
 statement1
 statement2
 statement3
       #Return statement is optional:
 return <value>
}
```

Example: See above.

See Also:

Name/Symbol: `DegToRad()`

Description: Converts degrees to radians.

Usage: `DegToRad(<deg>)`

Example: `DegToRad(180) # == 3.14159...`

See Also: `Cos()`, `Sin()`, `Tan()`, `ATan()`, `ACos()`, `ATan()`

Name/Symbol: `DesignBalancedSampling()`

Description: Samples elements "roughly" equally. This function returns a list of repeated samples from `<treatment_list>`, such that each element in `<treatment_list>` appears approximately equally. Each element from `<treatment_list>` is sampled once without replacement before all elements are returned to the mix and sampling is repeated. If there are no repeated items in `<list>`, there will be no consecutive repeats in the output. The last repeat-sampling will be truncated so that a `<length>`-size list is returned. If you don't want the repeated epochs this function provides, Shuffle() the results.

Usage:
```
DesignBalancedSampling(<list>, <length>)
```

Example:
```
DesignBalancedSampling([1,2,3,4,5],12)
## e.g., produces something like:
##    [5,3,1,4,2, 3,1,5,2,4, 3,1 ]
```

See Also: `CrossFactorWithoutDuplicates()`, `Shuffle()`, `DesignFullCounterBalance()`, `DesignGrecoLatinSquare()`, `DesignLatinSquare()`, `Repeat()`, `RepeatList()`, `LatinSquare()`

Name/Symbol: `DesignFullCounterbalance()`

Description: This takes two lists as parameters, and returns a nested list of lists that includes the full counterbalancing of both parameter lists. Use cautiously; this gets very large.

Usage:
```
DesignFullCounterbalance(<lista>, <listb>)
```

Example:
```
a <- [1,2,3]
b <- [9,8,7]
DesignFullCounterbalance(a,b) # == [[1,9],[1,8],[1,7],
#     [2,9],[2,8],[2,7],
#     [3,9],[3,8],[3,7]]
```

See Also: `CrossFactorWithoutDuplicates()`, `LatinSquare()`, `Shuffle()`, `DesignBalancedSampling()`, `DesignGrecoLatinSquare()`, `DesignLatinSquare()`, `Repeat()`, `RepeatList()`,

Name/Symbol: `DesignGrecoLatinSquare()`

Description: This will return a list of lists formed by rotating through each element of the `<treatment_list>`s, making a list containing all element of the list, according to a greco-latin square. All lists must be of the same length.

Usage: `DesignGrecoLatinSquare(<factor_list>, <treatment_list>, <treatment_list>)`

Example:
```
x <- ["a","b","c"]
y <- ["p","q","r"]
z <- ["x","y","z"]
Print(DesignGrecoLatinSquare(x,y,z))
# produces:    [[[a, p, x], [b, q, y], [c, r, z]],
#               [[a, q, z], [b, r, x], [c, p, y]],
#               [[a, r, y], [b, p, z], [c, q, x]]]
```

See Also: `CrossFactorWithoutDuplicates()`, `LatinSquare()` `DesignFullCounterBalance()`, `DesignBalancedSampling()`, `DesignLatinSquare()`, `Repeat()`, `RepeatList()`, `Shuffle()`

Name/Symbol: `DesignLatinSquare()`

Description: This returns return a list of lists formed by rotating through each element of `<treatment_list>`, making a list containing all element of the list. Has no side effect on input lists.

Usage: `DesignLatinSquare(<treatment1_list>, <treatment2_list>)`

Example:
```
order <- [1,2,3]
treatment <- ["A","B","C"]
design <- DesignLatinSquare(order,treatment)
# produces: [[[1, A], [2, B], [3, C]],
#            [[1, B], [2, C], [3, A]],
#            [[1, C], [2, A], [3, B]]]
```

See Also: `CrossFactorWithoutDuplicates()`, `DesignFullCounterBalance()`, `DesignBalancedSampling()`, `DesignGrecoLatinSquare()`, `Repeat()`, `LatinSquare()` `RepeatList()`, `Shuffle()`, `Rotate()`

Name/Symbol: `Div()`

Description: Returns round(`<num>/<mod>`)

Usage: `Div(<num>, <mod>)`

Example:

See Also: `Mod()`

Name/Symbol: `Draw()`

Description: Redraws the screen or a specific widget.

Usage:
```
Draw()
Draw(<object>)
```

Example:

See Also: `DrawFor()`, `Show()`, `Hide()`

Name/Symbol: `DrawFor()`

Description: Draws a screen or widget, returning after `<cycles>` refreshes. This function currently does not work as intended in the SDL implementation, because of a lack of control over the refresh blank. It may work in the future.

Usage: `DrawFor( <object>, <cycles>)`

Example:

See Also: `Draw()`, `Show()`, `Hide()`

6.6 E

Name/Symbol: `EasyLabel()`

Description: Creates and adds to the window location a label at specified location. Uses standard vera font with grey background. (May in the future get background color from window). Easy-to-use replacement for the `MakeFont`, `MakeLabel`, `AddObject`, `Move`, steps you typically have to go through.

Usage: `EasyLabel( <text>,<x>, <y>, <win>, <fontsize>)`

Example:
```
win <- MakeWindow()
lab <- EasyLabel("What?",200,100,win,12)
Draw()
```

See Also: `EasyTextBox()`, `MakeLabel()`

Name/Symbol: `EasyTextBox()`

Description: Creates and adds to the window location a textbox at specified location. Uses standard vera font with white background. Easy-to-use replacement for the MakeFont,MakeTextBox, AddObject, Move, steps.

Usage: `EasyTextBox( <text>,<x>, <y>, <win>, <fontsize>,<width>,<height>)`

Example:

```
win <- MakeWindow()
entry <- EasyTextBox("1 2 3 4 5",200,100,win,12,200,50)
Draw()
```

See Also: `EasyLabel()`, `MakeTextBox()`

Name/Symbol: `Ellipse()`

Description: Creates a ellipse for graphing at x,y with radii rx and ry. Ellipses are only currently definable oriented in horizontal/vertical directions. Ellipses must be added to a parent widget before it can be drawn; it may be added to widgets other than a base window. The properties of ellipses may be changed by accessing their properties directly, including the FILLED property which makes the object an outline versus a filled shape.

Usage: `Ellipse(<x>, <y>, <rx>, <ry>,<color>)`

Example:

```
e <- Ellipse(30,30,20,10, MakeColor(green))
AddObject(e, win)
Draw()
```

See Also: `Square()`, `Circle()`, `Rectangle()`, `Line()`

Name/Symbol: `EndOfFile()`

Description: Returns true if at the end of a file.

Usage: `EndOfFile(<filestream>)`

Example:

```
while(not EndOfFile(fstream))
{
 Print(FileReadLine(fstream))
}
```

See Also:

Name/Symbol: `EndOfLine()`

Description: Returns true if at end of line.

Usage: `EndOfLine(<filestream>)`

Example:

See Also:

Name/Symbol: `Exp()`

Description: e to the power of `<pow>`.

Usage: `Exp(<pow>)`

Example:
```
Exp(0)  # == 1
Exp(3) # == 20.0855
```

See Also: `Log()`

Name/Symbol: `ExtractListItems()`

Description: Extracts items from a list, forming a new list. The list `<items>` are the integers representing the indices that should be extracted.

Usage: `ExtractListItems(<list>,<items>)`

Example:
```
myList <- Sequence(101, 110, 1)
ExtractListItems(myList, [2,4,5,1,4])
# produces [102, 104, 105, 101, 104]
```

See Also: `Subset()`, `SubList()`, `SampleN()`

6.7 F

Name/Symbol: `FileClose()`

Description: Closes a filestream variable. Be sure to pass the variable name, not the filename.

Usage: `FileClose(<filestream>)`

Example:
```
x <- FileOpenRead("file.txt")
# Do relevant stuff here.
FileClose(x)
```

See Also: `FileOpenAppend()`, `FileOpenRead()`, `FileOpenWrite()`

Name/Symbol: `FileOpenAppend()`

Description: Opens a filename, returning a stream that can be used for writing information. Appends if the file already exists.

Usage: `FileOpenAppend(<filename>)`

Example:

See Also: `FileClose()`, `FileOpenRead()`, `FileOpenWrite()`

Name/Symbol: `FileOpenRead()`

Description: Opens a filename, returning a stream to be used for reading information.

Usage: `FileOpenRead(<filename>)`

Example:

See Also: `FileClose()`, `FileOpenAppend()`, `FileOpenWrite()`

Name/Symbol: `FileOpenWrite()`

Description: Opens a filename, returning a stream that can be used for writing information. Overwrites if file already exists.

Usage: `FileOpenWrite(<filename>)`

Example:

See Also: `FileClose()`, `FileOpenAppend()`, `FileOpenRead()`

Name/Symbol: `FilePrint()`

Description: Like `Print`, but to a file. Prints a string to a file, with a carriage return at the end. Returns a copy of the string it prints.

Usage: `FilePrint(<filestream>, <value>)`

Example: `FilePrint(fstream, "Another Line.")`

See Also: `Print()`, `FilePrint_()`

Name/Symbol: `FilePrint_()`

Description: Like `Print_`, but to a file. Prints a string to a file, without appending a newline character. Returns a copy of the string it prints.

Usage: `FilePrint_(<filestream>, <value>)`

Example: `FilePrint_(fstream, "This line doesn't end.")`

See Also: `Print_()`, `FilePrint()`

Name/Symbol: `FilePrintList()`

Description: Prints a list to a file, without the ','s or [] characters. Puts a carriage return at the end. Returns a string that was printed. If a list contains other lists, the printing will wrap multiple lines and the internal lists will be printed as normal. To avoid this, try FilePrintList(file,Flatten(list)).

Usage: `FilePrintList(<filestream>, <list>)`

Example:

```
FilePrintList(fstream, [1,2,3,4,5,5,5])
##
##  Produces:
##1 2 3 4 5 5 5
FilePrintList(fstream,[[1,2],[3,4],[5,6]])
#Produces:
# [1,2]
#,[3,4]
#,[5,6]
```

```
FilePrintList(fstream,Flatten([[1,2],[3,4],[5,6]]))
#Produces:
# 1 2 3 4 5 6
```

See Also: `Print()`, `Print_()`, `FilePrint()`, `FilePrint_()`, `PrintList()`,

Name/Symbol: `FileReadCharacter()`

Description: Reads and returns a single character from a filestream.

Usage: `FileReadCharacter(<filestream>)`

Example:

See Also:

Name/Symbol: `FileReadLine()`

Description: Reads and returns a line from a file; all characters up until the next newline or the end of the file.

Usage: `FileReadLine(<filestream>)`

Example:

See Also:

Name/Symbol: `FileReadList()`

Description: Given a filename, will open it, read in all the items into a list (one item per line), and close the file afterward. Ignores blank lines or lines starting with `#`. Useful with a number of pre-defined data files stored in `media/text/`. See Section 4.18: Provided Media Files.

Usage: `FileReadList(<filename>)`

Example: `FileReadList("data.txt")`

See Also:

Name/Symbol: `FileReadTable()`

Description: Reads a table directly from a file. Data in file should separated by spaces. Reads each line onto a sublist, with space-separated tokens as items in sublist. Ignores blank lines or lines beginning with `#`. Optionally, specify a token separator other than space.

Usage: `FileReadTable(<filename>, <optional-separator>)`

Example: `a <- FileReadTable("data.txt")`

See Also: `FileReadList()`

Name/Symbol: `FileReadText()`

Description: Returns all of the text from a file, ignoring any lines beginning with `#`. Opens and closes the file transparently.

Usage: `FileReadText(<filename>)`

Example: `instructions <- FileReadText("instructions.txt")`

See Also: `FileReadList()`, `FileReadTable()`

Name/Symbol: `FileReadWord()`

Description: Reads and returns a 'word' from a file; the next connected stream of characters not including a `' '` or a newline. Will not read newline characters.

Usage: `FileReadWord(<filestream>)`

Example:

See Also: `FileReadLine()`, `FileReadTable()`, `FileReadList()`

Name/Symbol: `FindInString()`

Description: Finds a token in a string, returning the position.

Usage: `FindInString(<string>,<string>)`

Example: `FindInString("about","bo")  # == 2`

See Also: `SplitString()`

Name/Symbol: `First()`

Description: Returns the first item of a list.

Usage: `First(<list>)`

Example:
```
First([3,33,132]) # == 3
```

See Also: `Nth()`, `Last()`

Name/Symbol: `Flatten()`

Description: Flattens nested list `<list>` to a single flat list.

Usage: `Flatten(<list>)`

Example:
```
Flatten([1,2,[3,4],[5,[6,7],8],[9]]) # == [1,2,3,4,5,6,7,8,9]
Flatten([1,2,[3,4],[5,[6,7],8],[9]]) # == [1,2,3,4,5,6,7,8,9]
```

See Also: `FlattenN()`, `FoldList()`

Name/Symbol: `FlattenN()`

Description: Flattens `<n>` levels of nested list `<list>`.

Usage: `Flatten(<list>, <n>)`

Example:
```
Flatten([1,2,[3,4],[5,[6,7],8],[9]],1)
# == [1,2,3,4,5,[6,7],8,9]
```

See Also: `Flatten()`, `FoldList()`

Name/Symbol: `Floor()`

Description: Rounds `<num>` down to the next integer.

Usage: `Floor(<num>)`

Example:
```
Floor(33.23) # == 33
Floor(3.999)   # ==3
Floor(-32.23)  # == -33
```

See Also: `AbsFloor()`, `Round()`, `Ceiling()`

Name/Symbol: `FoldList()`

Description: Folds a list into equal-length sublists.

Usage: `FoldList(<list>, <size>)`

Example: `FoldList([1,2,3,4,5,6,7,8],2) # == [[1,2],[3,4],[5,6],[7,8]]`

See Also: `FlattenN()`, `Flatten()`

Name/Symbol: `Format()`

Description: Formats the printing of values to ensure the proper spacing. It will either truncate or pad `<value>` with spaces so that it ends up exactly `<length>` characters long. Character padding is at the end.

Usage: `Format(<value>, <length>)`

Example:

```
x <- 33.23425225
y <- 23.3
Print("["+Format(x,5)+"]")
Print("["+Format(y,5)+"]")
## Output:
## [33.23 ]
## [23.3  ]
```

See Also: `CR()` `Tab()`

6.8 G

Name/Symbol: `GetCursorPosition()`

Description: Returns an integer specifying where in a textbox the edit cursor is. The value indicates which character it is on.

Usage: `GetCursorPosition(<textbox>)`

Example:

See Also: `SetCursorPosition()`, `MakeTextBox()`, `SetText()`

Name/Symbol: `GetData()`

Description: Gets Data from network connection. Example of usage in demo/nim.pbl.

Usage: `val <- GetData(<network>,<size>)`

Example: On 'server':

```
net <- WaitForNetworkConnection("localhost",1234)
SendData(net,"Watson, come here. I need you.")
value <-  GetData(net,10)
Print(value)
```

On Client:

```
net <- ConnectToHost("localhost",1234)
value <-  GetData(net,20)
Print(value)
##should print out "Watson, come here. I need you."
```

See Also: `ConnectToIP`, `ConnectToHost`, `WaitForNetworkConnection`, `SendData`, `ConvertIPString`, `CloseNetworkConnection`

Name/Symbol: `GetInput()`

Description: Allows user to type input into a textbox.

Usage: `GetInput(<textbox>,<escape-key>)`

Example:

See Also: `SetEditable()`, `GetCursorPosition()`, `MakeTextBox()`, `SetText()`

Name/Symbol: `GetMouseCursorPosition()`

Description: Gets the current x,y coordinates of the mouse pointer.

Usage: `GetMouseCursorPosition()`

Example:

```
pos <- GetMouseCursorPosition()
```

See Also: `ShowCursor`, `WaitForMouseButton`, `SetMouseCursorPosition`, `GetMouseCursorPosition`

Name/Symbol: `GetNIMHDemographics()`

Description: Gets demographic information that are normally required for NIMH-related research. Currently are gender (M/F/prefer not to say), ethnicity (Hispanic or not), and race (A.I./Alaskan, Asian/A.A., Hawaiian, black/A.A., white/Caucasian, other). It then prints their responses in a single line in the demographics file, along with any special code you supply and a time/date stamp. This code might include a subject number, experiment number, or something else, but many informed consent forms assure the subject that this information cannot be tied back to them or their data, so be careful about what you record. The file output will look something like:

```
----
x0413 Thu Apr 22 17:58:15 2004 1 Y 4
x0413 Thu Apr 23 17:58:20 2004 3 Y 5
x0413 Thu Apr 24 12:41:30 2004 2 Y 5
x0413 Thu Apr 24 14:11:54 2004 2 N 5
----
```

The first column is the user-specified code (in this case, indicating the experiment number). The middle columns indicate date/time, and the last three columns indicate gender (M, F, other), Hispanic (Y/N), and race.

Usage: `GetNIMHDemographics(<code-to-print-out>, <window>, <filename>)`

Example: `GetNIMHDemographics("x0413", gwindow, "x0413-demographics.txt")`

See Also:

Name/Symbol: `GetPEBLVersion()`

Description: Returns a string describing which version of PEBL you are running.

Usage: `GetPEBLVersion()`

Example: `Print(GetPEBLVersion())`

See Also: `TimeStamp()`

Name/Symbol: `GetSize()`

Description: Returns a list of `[height, width]`, specifying the size of the widget. The .width and .height properties can also be used instead of this function

Usage: `GetSize(<widget>)`

Example:
```
image <- MakeImage("stim1.bmp")
xy <- GetSize(image)
x <- Nth(xy, 1)
y <- Nth(xy, 2)
```

See Also:

Name/Symbol: `GetSubNum()`

Description: Creates dialog to ask user to input a subject code

Usage: `GetSubNum(<win>)`

Example:
```
## Put this at the beginning of an experiment,
## after a window gWin has been defined.
##
 if(gSubNum == 0)
  {
    gSubNum <- GetSubNum(gWin)
  }
```

Note: gSubNum can also be set from the command line.

See Also:

Name/Symbol: `GetText()`

Description: Returns the text stored in a text object (either a textbox or a label). The .text properties can also be used instead of this function.

Usage: `GetText(<widget>)`

Example:

See Also: `SetCursorPosition()`, `GetCursorPosition()`, `SetEditable()`, `MakeTextBox()`

Name/Symbol: `GetTime()`

Description: Gets time, in milliseconds, from when PEBL was initialized. Do not use as a seed for the RNG, because it will tend to be about the same on each run. Instead, use `RandomizeTimer()`.

Usage: `GetTime()`

Example:
```
a <- GetTime()
WaitForKeyDown("A")
b <- GetTime()
Print("Response time is: " + (b - a))
```

See Also: `TimeStamp()`

6.9 H

Name/Symbol: `Hide()`

Description: Makes an object invisible, so it will not be drawn.

Usage: `Hide(<object>)`

Example:
```
window <- MakeWindow()
image1  <- MakeImage("pebl.bmp")
image2  <- MakeImage("pebl.bmp")
AddObject(image1, window)
AddObject(image2, window)
Hide(image1)
Hide(image2)
Draw() # empty screen will be drawn.

Wait(3000)
Show(image2)
Draw() # image2 will appear.

Hide(image2)
Draw() # image2 will disappear.

Wait(1000)
Show(image1)
Draw() # image1 will appear.
```

See Also: `Show()`

6.10 I

Name/Symbol: `if`

Description: Simple conditional test.

Usage:
```
if(test)
{
 statements
 to
 be
 executed
}
```

Example:

See Also:

Name/Symbol: `if...elseif...else`

Description: Complex conditional test. Be careful of spacing the else—if you put carriage returns on either side of it, you will get a syntax error. The `elseif` is optional, but multiple `elseif` statements can be strung together. The `else` is also optional, although only one can appear.

Usage:
```
if(test)
{
 statements if true
} elseif (newtest) {
 statements if newtest true; test false
} else {
 other statements
}
```

Example:
```
 if(3 == 1) {
             Print("ONE")
  }elseif(3==4){
             Print("TWO")
  }elseif(4==4){
             Print("THREE")
  }elseif(4==4){
             Print("FOUR")
  }else{Print("FIVE")}
```

See Also: if

Name/Symbol: `IsAnyKeyDown()`

Description:

Usage:

Example:

See Also:

Name/Symbol: `IsAudioOut()`

Description: Tests whether `<variant>` is a AudioOut stream.

Usage: `IsAudioOut(<variant>)`

Example:
```
if(IsAudioOut(x))
{
 Play(x)
}
```

See Also: `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsColor()`

Description: Tests whether `<variant>` is a Color.

Usage: `IsColor(<variant>)`

Example:
```
if(IsColor(x)
{
 gWin <- MakeWindow(x)
}
```

See Also: `IsAudioOut()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`, `IsWindow()`

Name/Symbol: `IsImage()`

Description: Tests whether `<variant>` is an Image.

Usage: `IsImage(<variant>)`

Example:
```
if(IsImage(x))
{
 AddObject(gWin, x)
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsInteger()`

Description: Tests whether `<variant>` is an integer type. Note: a number represented internally as a floating-point type whose is an integer will return false. Floating-point numbers can be converted to internally- represented integers with the `ToInteger()` or `Round()` commands.

Usage: `IsInteger(<variant>)`

Example:
```
x <- 44
y <- 23.5
z <- 6.5
test <- x + y + z

IsInteger(x) # true
IsInteger(y) # false
IsInteger(z) # false
IsInteger(test) # false
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsFileStream()`

Description: Tests whether `<variant>` is a FileStream object.

Usage: `IsFileStream(<variant>)`

Example:
```
if(IsFileStream(x))
{
 Print(FileReadWord(x)
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsFloat()`

Description: Tests whether `<variant>` is a floating-point value. Note that floating-point can represent integers with great precision, so that a number appearing as an integer can still be a float.

Usage:
```
IsFloat(<variant>)
```

Example:
```
x <- 44
y <- 23.5
z <- 6.5
test <- x + y + z

IsFloat(x)      # false
IsFloat(y)      # true
IsFloat(z)      # true
IsFloat(test)   # true
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsFont()`

Description: Tests whether `<variant>` is a Font object.

Usage:
```
IsFont(<variant>)
```

Example:
```
if(IsFont(x))
{
 y <- MakeLabel("stimulus", x)
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsKeyDown()`

Description:

Usage:

Example:

See Also: `IsKeyUp()`

Name/Symbol: `IsKeyUp()`

Description:

Usage:

Example:

See Also: `IsKeyDown()`

Name/Symbol: `IsLabel()`

Description: Tests whether `<variant>` is a text Label object.

Usage: `IsLabel(<variant>)`

Example:
```
if(IsLabel(x)
{
 text <- GetText(x)
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsList()`

Description: Tests whether `<variant>` is a PEBL list.

Usage: `IsList(<variant>)`

Example:
```
if(IsList(x))
{
 loop(item, x)
 {
  Print(item)
 }
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsMember()`

Description: Returns true if `<element>` is a member of `<list>`.

Usage: `IsMember(<element>,<list>)`

Example:
```
IsMember(2,[1,4,6,7,7,7,7]) # false
IsMember(2,[1,4,6,7,2,7,7,7])  # true
```

See Also:

Name/Symbol: `IsNumber()`

Description: Tests whether `<variant>` is a number, either a floating-point or an integer.

Usage: `IsNumber(<variant>)`

Example:
```
if(IsNumber(x))
{
 Print(Sequence(x, x+10, 1))
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsString()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsShape`

Description: Tests whether `<variant>` is a drawable shape, such as a circle, square rectangle, line, bezier curve, or polygon.

Usage: `IsShape(<variant>)`

Example:
```
if(IsShape(x))
{
  Move(x,300,300)
}
```

See Also: `Square()`, `Circle()`, `Rectangle()`, `Line()`, `Bezier()`, `Polygon()` `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`, `IsWindow()`

Name/Symbol: `IsString()`

Description: Tests whether `<variant>` is a text string.

Usage: `IsString(<variant>)`

Example:
```
if(IsString(x))
{
 tb <- MakeTextBox(x, 100, 100)
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsTextBox()`, `IsWidget()`

Name/Symbol: `IsTextBox()`

Description: Tests whether `<variant>` is a TextBox Object

Usage: `IsTextBox(<variant>)`

Example:
```
if(IsTextBox(x))
{
 Print(GetText(x))
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsWidget()`

Name/Symbol: `IsWidget`

Description: Tests whether `<variant>` is any kind of a widget object (image, label, or textbox).

Usage: `IsWidget(<variant>)`

Example:
```
if(IsWidget(x))
{
 Move(x, 200,300)
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`

Name/Symbol: `IsWindow`

Description: Tests whether `<variant>` is a window.

Usage: `IsWindow(<variant>)`

Example:
```
if(IsWindow(x))
{
  AddObject(y,x)
}
```

See Also: `IsAudioOut()`, `IsColor()`, `IsImage()`, `IsInteger()`, `IsFileStream()`, `IsFloat()`, `IsFont()`, `IsLabel()`, `IsList()`, `IsNumber()`, `IsString()`, `IsTextBox()`

6.11 L

Name/Symbol: `Last()`

Description: Returns the last item in a list. Provides faster access to the last item of a list than does Nth().

Usage: `Last(<list>)`

Example: `Last([1,2,3,444]) # == 444`

See Also: `Nth()`, `First()`

Name/Symbol: `LatinSquare()`

Description: Quick and dirty latin square, taking on just one list argument.

Usage: `LatinSquare(<list>)`

Example:
```
Print(LatinSquare([11,12,13,14,15,16]))
# Output:
#[[11, 12, 13, 14, 15, 16]
#, [12, 13, 14, 15, 16, 11]
#, [13, 14, 15, 16, 11, 12]
#, [14, 15, 16, 11, 12, 13]
#, [15, 16, 11, 12, 13, 14]
#, [16, 11, 12, 13, 14, 15]
#]
```

See Also: `DesignFullCounterBalance()`, `DesignBalancedSampling()`, `DesignGrecoLatinSquare()`, `DesignLatinSquare()`, `Repeat()`, `RepeatList()`, `Shuffle()`

Name/Symbol: `Line()`

Description: Creates a line for graphing at x,y ending at x+dx, y+dy. dx and dy describe the size of the line. Lines must be added to a parent widget before it can be drawn; it may be added to widgets other than a base window. Properties of lines may be accessed and set later.

Usage: `Line(<x>, <y>, <dx>, <dy>, <color>)`

Example:
```
l <- Line(30,30,20,20, MakeColor("green")
AddObject(l, win)
Draw()
```

See Also: `Square()`, `Ellipse()`, `Rectangle()`, `Circle()`

Name/Symbol: `List()`

Description: Creates a list of items. Functional version of `[]`.

Usage: `List(<item1>, <item2>, ....)`

Example: `List(1,2,3,444) # == [1,2,3,444]`

See Also: `[ ]`, `Merge()`, `Append()`

Name/Symbol: `ListToString()`

Description: Converts a list of things to a single string

Usage: `ListToString(<list>)`

Example:
```
ListToString([1,2,3,444]) # == "123444"
ListToString(["a","b","c","d","e"]) # == "abcde"
```

See Also: `SubString`, `StringLength`

Name/Symbol: `Length()`

Description: Returns the number of items in a list.

Usage: `Length(<list>)`

Example: `Length([1,3,55,1515]) # == 4`

See Also: `StringLength()`

Name/Symbol: `LoadSound()`

Description: Loads a soundfile from `<filename>`, returning a variable that can be played.

Usage: `LoadSound(<filename>)`

Example:

See Also:

Name/Symbol: `Log10()`

Description: Log base 10 of `<num>`.

Usage: `Log10(<num>)`

Example:

See Also: `Log2()`, `LogN()`, `Ln()`, `Exp()`

Name/Symbol: `Log2()`

Description: Log base 2 of `<num>`.

Usage: `Log2(<num>)`

Example:

See Also: `Log()`, `LogN()`, `Ln()`, `Exp()`

Name/Symbol: `LogN()`

Description: Log base `<base>` of `<num>`.

Usage: `LogN(<num>, <base>)`

Example:
```
LogN(100,10) # == 2
LogN(256,2) # == 8
```

See Also: `Log()`, `Log2()`, `Ln()`, `Exp()`

Name/Symbol: `Lowercase()`

Description: Changes a string to lowercase. Useful for testing user input against a stored value, to ensure case differences are not detected.

Usage: `Lowercase(<string>)`

Example: `Lowercase("POtaTo") # == "potato"`

See Also: `Uppercase()`

Name/Symbol: `Ln()`

Description: Natural log of `<num>`.

Usage: `Ln(<num>)`

Example:

See Also: `Log()`, `Log2()`, `LogN()`, `Exp()`

Name/Symbol: `Lookup()`

Description: Returns element in `<database>` corresponding to element of `<keylist>` that matches `<key>`.

Usage: `Lookup(<key>,<keylist>,<database>)`

Example:
```
 keys     <- [1,2,3,4,5]
 database <- ["market","home","roast beef","none","wee wee wee"]
 Print(Lookup(3,keys,database)))

## Or, do something like this:

data  <- [["punky","brewster"],
          ["arnold","jackson"],
          ["richie","cunningham"],
          ["alex","keaton"]]
```

```
d2 <- Transpose(data)
key <- First(data)

Print(Lookup("alex", key, data))
##Returns ["alex","keaton"]
```

See Also:

Name/Symbol: `loop()`

Description: Loops over elements in a list. During each iteration, `<counter>` is bound to each consecutive member of `<list>`.

Usage:
```
loop(<counter>, <list>)
{
 statements
 to
 be
 executed
}
```

Example:

See Also: `while()`, `{ }`

6.12 M

Name/Symbol: `MakeColor()`

Description: Makes a color from `<colorname>` such as "red", "green", and nearly 800 others. Color names and corresponding RGB values can be found in `doc/colors.txt`.

Usage: `MakeColor(<colorname>)`

Example:

See Also: `MakeColorRGB()`

Name/Symbol: `MakeColorRGB()`

Description: Makes an RGB color by specifying `<red>`, `<green>`, and `<blue>` values (between 0 and 255).

Usage: `MakeColorRGB(<red>, <green>, <blue>)`

Example:

See Also: `MakeColor()`

Name/Symbol: `MakeFont()`

Description: Makes a font.

Usage: `MakeFont(<ttf_filename>, <style>, <size>, <fgcolor>, <bgcolor>, <anti-aliased>)`

Example:

See Also:

Name/Symbol: `MakeImage()`

Description: Makes an image widget from an image file. `.bmp` formats should be supported; others may be as well.

Usage:
`MakeImage(<filename>)`

Example:

See Also:

Name/Symbol: `MakeLabel()`

Description: Makes a text label for display on-screen. Text will be on a single line, and the `Move()` command centers `<text>` on the specified point.

Usage: `MakeLabel(<text>, <font>)`

Example:

See Also:

Name/Symbol: `MakeNGonPoints()`

Description: Creates a set of points that form a regular n-gon. It can be transformed with functions like `RotatePoints`, or it can be used to create a graphical object with `Polygon`.

Note: MakeNGonPoints returns a list [[x1, x2, x3,...],[y1,y2,y3,...]], while Polygon() takes the X and Y lists independently.

Usage: `MakeNGonPoints(<radius>, <num_peaks>)`

Example:

```
window <- MakeWindow()
ngonp <- MakeNGonPoints(50,10)
ngon <- PolyGon(200,200,First(ngonp),Nth(ngonp,2),MakeColor("red"),1)
AddObject(ngon,window)
Draw()
```

See Also: `MakeStarPoints`, `Polygon`, `RotatePoints`, `ZoomPoints`

Name/Symbol: `MakeStarPoints()`

Description: Creates a set of points that form a regular star. It can be transformed with functions like `RotatePoints`, or it can be used to create a graphical object with `Polygon`.

Note: MakeStarPoints returns a list [[x1, x2, x3,...],[y1,y2,y3,...]], while Polygon() takes the X and Y lists independently.

Usage: `MakeNGonPoints(<outer_radius>, <inner_radius>, <num_peaks>)`

Example:

```
window <- MakeWindow()
sp <- MakeStarPoints(50,20,10)
star <- PolyGon(200,200,First(sp),Nth(sp,2),MakeColor("red"),1)
AddObject(star,window)
Draw()
```

See Also: `MakeNGonPoints`, `Polygon`, `RotatePoints`, `ZoomPoints`

Name/Symbol: `MakeTextBox()`

Description: Creates a textbox in which to display text. Textboxes allow multiple lines of text to be rendered; automatically breaking the text into lines.

Usage: `MakeWindow(<text>,<font>,<width>,<height>)`

Example:

```
font <-MakeFont("Vera.ttf", 1, 12, MakeColor("red"),
MakeColor("green"), 1)
tb <- MakeTextBox("This is the text in the textbox",
font, 100, 250)
```

See Also: `MakeLabel()`, `GetText()`, `SetText()`, `SetCursorPosition()`, `GetCursorPosition()`, `SetEditable()`

Name/Symbol: `MakeWindow()`

Description: Creates a window to display things in. Background is specified by `<color>`.

Usage: `MakeWindow(<color>)`

Example:

See Also:

Name/Symbol: `Max()`

Description: Returns the largest of `<list>`.

Usage: `Max(<list>)`

Example:
```
c <- [3,4,5,6]
m <- Max(c) # m == 6
```

See Also: `Min()`, `Mean()`, `StDev()`

Name/Symbol: `Mean()`

Description: Returns the mean of the numbers in `<list>`.

Usage: Mean(`<list-of-numbers>`)

Example:
```
c <- [3,4,5,6]
m <- Mean(c) # m == 4.5
```

See Also: `Median()`, `Quantile()`, `StDev()`, `Min()`, `Max()`

Name/Symbol: `Median()`

Description: Returns the median of the numbers in `<list>`.

Usage: Median(`<list-of-numbers>`)

Example:
```
c <- [3,4,5,6,7]
m <- Median(c) # m == 5
```

See Also: `Mean()`, `Quantile()`, `StDev()`, `Min()`, `Max()`

Name/Symbol: `Merge()`

Description: Combines two lists, `<lista>` and `<listb>`, into a single list.

Usage: `Merge(<lista>,<listb>)`

Example:
```
Merge([1,2,3],[8,9])  # == [1,2,3,8,9]
```

See Also: `[ ]`, `Append()`, `List()`

Name/Symbol: `Min()`

Description: Returns the 'smallest' element of a list.

Usage: `Min(<list>)`

Example:
```
  c <- [3,4,5,6]
  m <-  Min(c) # == 3
```

See Also: `Max()`

Name/Symbol: `Mod()`

Description: Returns `<num>`, `<mod>`, or remainder of `<num>/<mod>`

Usage: `Mod( <num> <mod>)`

Example:
```
Mod(34, 10) # == 4
Mod(3, 10) # == 3
```

See Also: `Div()`

Name/Symbol: `Move()`

Description: Moves an object to a specified location. Images and Labels are moved according to their center; TextBoxes are moved according to their upper left corner.

Usage: `Move(<object>, <x>, <y>)`

Example:
```
Move(label, 33, 100)
```

See Also: `MoveCorner()`, `MoveCenter()`, `.X` and `.Y` properties.

Name/Symbol: `MoveCorner()`

Description: Moves a label or image to a specified location according to its upper left corner, instead of its center.

Usage: `MoveCorner(<object>, <x>, <y>)`

Example: `MoveCorner(label, 33, 100)`

See Also: `Move()`, `MoveCenter()`, `.X` and `.Y` properties

Name/Symbol: `MoveCenter()`

Description: Moves a TextBox to a specified location according to its center, instead of its upper left corner.

Usage: `MoveCenter(<object>, <x>, <y>)`

Example: `MoveCenter(TextBox, 33, 100)`

See Also: `Move()`, `MoveCorner()`, `.X` and `.Y` properties

Name/Symbol: `not`

Description: Logical not

Usage:

Example:

See Also: `and`, `or`

6.13 N

Name/Symbol: `NormalDensity()`

Description: Computes density of normal standard distribution

Usage: `NormalDensity(<x>)`

Example:

```
Print(NormalDensity(-100))     # 1.8391e-2171
Print(NormalDensity(-2.32635)) #5.97
Print(NormalDensity(0))        #0.398942
Print(NormalDensity(1.28155))  #.90687
Print(NormalDensity(1000))     #inf
```

See Also: `RandomNormal()`, `CumNormInv()`

Name/Symbol: `Nth()`

Description: Extracts the Nth item from a list. Indexes from 1 upwards. `Last()` provides faster access than `Nth()` to the end of a list, which must walk along the list to the desired position.

Usage: `Nth(<list>, <index>)`

Example:

```
a <- ["a","b","c","d"]
Print(Nth(a,3))  # == 'c'
```

See Also: `First()`, `Last()`

Name/Symbol: `NthRoot()`

Description: `<num>` to the power of 1/`<root>`.

Usage: `NthRoot(<num>, <root>)`

Example:

See Also:

6.14 O

Name/Symbol: `or`

Description: Logical or

Usage:

Example:

See Also: `and`, `not`

Name/Symbol: `Order()`

Description: Returns a list of indices describing the order of values by position, from min to max.

Usage: `Order(<list-of-numbers>)`

Example:
```
n <- [33,12,1,5,9]
   o <- Order(n)
    Print(o) #should print [3,4,5,2,1]
```

See Also: `Rank()`

6.15 P

Name/Symbol: `PlayForeground()`

Description: Plays the sound 'in the foreground'; does not return until the sound is complete.

Usage: `PlayForeground(<sound>)`

Example:

See Also: `PlayBackground()`, `Stop()`

Name/Symbol: `PlayBackground()`

Description: Plays the sound 'in the background', returning immediately.

Usage: `PlayBackground(<sound>)`

Example:

See Also: `PlayForeground()`, `Stop()`

Name/Symbol: `Plus`

Description: Creates a polygon in the shape of a plus sign. Arguments include position in window.

- `<x>` and `<y>` is the position of the center
- `<size>` or the size of the plus sign in pixels
- `<width>` thickness of the plus
- `<color>` is a color object (not just the name)

Like other drawn objects, the plus must then be added to the window to appear.

Usage:
```
Plus(x,y,size,width,color)
```

Example:
```
win <- MakeWindow()
p1 <- Plus(100,100,80,15,MakeColor("red"))
AddObject(p1,win)
Draw()
```

See Also: `BlockE()`, `Polygon()`, `MakeStarPoints()`, `MakeNGonPoints()`

Name/Symbol: `Polygon`

Description: Creates a polygon in the shape of the points specified by `<xpoints>`, `<ypoints>`. The lists `<xpoints>` and `<ypoints>` are adjusted by `<x>` and `<y>`, so they should be relative to 0, not the location you want the points to be at.

Like other drawn objects, the polygon must then be added to the window to appear.

Usage:
```
Polygon(<x>,<y>,<xpoints>,<ypoints>,<color>,<filled>)
```

Example:
```
win <- MakeWindow()
 #This makes a T
 xpoints <- [-10,10,10,20,20,-20,-20,-10]
 ypoints <- [-20,-20,40,40,50,50,40,40]
p1 <-    Polygon(100,100,xpoints, ypoints,MakeColor("black"),1)
AddObject(p1,win)
Draw()
```

See Also: `BlockE()`, `Bezier()`, `MakeStarPoints()`, `MakeNGonPoints()`

Name/Symbol: `Pow()`

Description: Raises or lowers `<num>` to the power of `<pow>`.

Usage:
```
Pow(<num>, <pow>)
```

Example:
```
Pow(2,6) # == 64
Pow(5,0) # == 1
```

See Also:

Name/Symbol: `Print()`

Description: Prints `<value>` to stdout (the console [Linux] or the file `stdout.txt` [Windows]), and then appends a newline afterwards.

Usage: `Print(<value>)`

Example:

```
Print("hello world")
Print(33 + 43)
x <-Print("Once")
```

See Also: `Print_()`, `FilePrint()`

Name/Symbol: `PrintProperties()`

Description: Prints .properties/values for any complex object. These include textboxes, fonts, colors, images, shapes, etc. Mostly useful as a debugging tool.

Usage: `PrintProperties(<object>)`

Example:

```
win <- MakeWindow()
tb <- EasyTextbox("one",20,20,win,22,400,80)
PrintProperties(tb)

##Output:
----------
[CURSORPOS]: 0
[EDITABLE]: 0
[HEIGHT]: 80
[ROTATION]: 0
[TEXT]: one
[VISIBLE]: 1
[WIDTH]: 400
[X]: 20
[Y]: 20
[ZOOMX]: 1
[ZOOMY]: 1
----------
```

See Also: `Print()`

Name/Symbol: `Print_()`

Description: Prints `<value>` to stdout; doesn't append a newline afterwards.

Usage: `Print_(<value>)`

Example:
```
Print_("This line")
Print_(" ")
Print_("and")
Print_(" ")
Print("Another line")
# prints out: 'This line and Another line'
```

See Also: `Print()`, `FilePrint()`

Name/Symbol: `PrintList()`

Description: Prints a list, without the ','s or [] characters. Puts a carriage return at the end. Returns a string that was printed. If a list contains other lists, the printing will wrap multiple lines and the internal lists will be printed as normal. To avoid this, try PrintList(file,Flatten(list)).

Usage: `PrintList(<filestream>, <list>)`

Example:
```
PrintList( [1,2,3,4,5,5,5])
##
##  Produces:
##1 2 3 4 5 5 5
PrintList([[1,2],[3,4],[5,6]])
#Produces:
# [1,2]
#,[3,4]
#,[5,6]

PrintList(Flatten([[1,2],[3,4],[5,6]]))
#Produces:
# 1 2 3 4 5 6
```

See Also: `Print()`, `Print_()`, `FilePrint()`, `FilePrint_()`, `FilePrintList()`,

6.16 Q

Name/Symbol: `Quantile()`

Description: Returns the `<num>` quantile of the numbers in `<list>`. `<num>` should be between 0 and 100

Usage: `Quantile(<list>, <num>)`

Example:
```
##Find 75th percentile to use as a threshold.
thresh <- Quantile(rts,75)
```

See Also: `StDev()`, `Median()`, `Mean()`, `Max()`, `Min()`

6.17 R

Name/Symbol: `RadToDeg()`

Description: Converts `<rad>` radians to degrees.

Usage: `RadToDeg( <rad>)`

Example:

See Also: `DegToRad()`, `Tan()`, `Cos()`, `Sin()`, `ATan()`, `ASin()`, `ACos()`

Name/Symbol: `Random()`

Description: Returns a random number between 0 and 1.

Usage: `Random()`

Example: `a <- Random()`

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial()`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogistic()`, `RandomLogNormal()`, `RandomNormal()`, `RandomUniform()`, `RandomizeTimer()`, `SeedRNG()`

Name/Symbol: `RandomBernoulli()`

Description: Returns 0 with probability `(1-<p>)` and 1 with probability `<p>`.

Usage: `RandomBernoulli(<p>)`

Example: `RandomBernoulli(.3)`

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogistic()`, `RandomLogNormal()`, `RandomNormal()`, `RandomUniform()`, `RandomizeTimer()`, `SeedRNG()`

Name/Symbol: `RandomBinomial`

Description: Returns a random number according to the Binomial distribution with probability `<p>` and repetitions `<n>`, i.e., the number of `<p>` Bernoulli trials that succeed out of `<n>` attempts.

Usage: `RandomBinomial(<p> <n>)`

Example:
```
RandomBinomial(.3, 10) # returns a number from 0 to 10
```

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogistic()`, `RandomLogNormal()`, `RandomNormal()`, `RandomUniform()`, `RandomizeTimer()`, `SeedRNG()`

Name/Symbol: `RandomDiscrete()`

Description: Returns a random integer between 1 and the argument (inclusive), each with equal probability. If the argument is a floating-point value, it will be truncated down; if it is less than 1, it will return 1, and possibly a warning message.

Usage: `RandomDiscrete(<num>)`

Example:
```
RandomDiscrete(30) # Returns a random integer between 1 and 30
```

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogistic()`, `RandomLogNormal()`, `RandomNormal()`, `RandomUniform()`, `RandomizeTimer()`, `SeedRNG()`

Name/Symbol: `RandomExponential()`

Description: Returns a random number according to exponential distribution with mean `<mean>` (or decay 1/mean).

Usage: `RandomExponential(<mean>)`

Example: `RandomExponential(100)`

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomLogistic()`, `RandomLogNormal()`, `RandomNormal()`, `RandomUniform()`, `RandomizeTimer`, `SeedRNG()`

Name/Symbol: `RandomizeTimer()`

Description: Seeds the RNG with the current time.

Usage: `RandomizeTimer()`

Example:
```
RandomizeTimer()
x <- Random()
```

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogistic()`, `RandomLogNormal()`, `RandomNormal()`, `RandomUniform()`, `SeedRNG()`

Name/Symbol: `RandomLogistic()`

Description: Returns a random number according to the logistic distribution with parameter `<p>`: f(x) = exp(x)/(1+exp(x))

Usage: `RandomLogistic(<p>)`

Example: RandomLogistic(.3)

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogNormal()`, `RandomNormal()`, `RandomUniform()`, `RandomizeTimer`, `SeedRNG()`

Name/Symbol: `RandomLogNormal()`

Description: Returns a random number according to the log-normal distribution with parameters `<median>` and `<spread>`. Generated by calculating median `*` exp(spread `*` RandomNormal(0,1)). `<spread>` is a shape parameter, and only affects the variance as a function of the median; similar to the coefficient of variation. A value near 0 is a sharp distribution (.1-.3), larger values are more spread out; values greater than 2 make little difference in the shape.

Usage: `RandomLogNormal(<median>, <spread>)`

Example: `RandomLogNormal(5000, .1)`

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogistic()`, `RandomNormal()`, `RandomUniform()`, `RandomizeTimer`, `SeedRNG()`

Name/Symbol: `RandomNormal()`

Description: Returns a random number according to the standard normal distribution with `<mean>` and `<stdev>`.

Usage: `RandomNormal(<mean>, <stdev>)`

Example:

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogistic()`, `RandomLogNormal()`, `RandomUniform()`, `RandomizeTimer`, `SeedRNG()`

Name/Symbol: `RandomUniform()`

Description: Returns a random floating-point number between 0 and `<num>`.

Usage: `RandomUniform(<num>)`

Example:

See Also: `Random()`, `RandomBernoulli()`, `RandomBinomial`, `RandomDiscrete()`, `RandomExponential()`, `RandomLogistic()`, `RandomLogNormal()`, `RandomNormal()`, `RandomizeTimer()`, `SeedRNG()`

Name/Symbol: `Rank()`

Description: Returns a list of numbers describing the rank of each position, from min to max. The same as calling `Order(Order(x))`.

Usage: `Rank(<list-of-numbers>)`

Example:
```
n <- [33,12,1,5,9]
   o <- Rank(n)
    Print(o) #should print [5,4,1,2,3]
```

See Also: `Order()`

Name/Symbol: `Rectangle()`

Description: Creates a rectangle for graphing at x,y with size dx and dy. Rectangles are only currently definable oriented in horizontal/vertical directions. A rectangle must be added to a parent widget before it can be drawn; it may be added to widgets other than a base window. The properties of rectangles may be changed by accessing their properties directly, including the FILLED property which makes the object an outline versus a filled shape.

Usage: `Rectangle(<x>, <y>, <dx>, <dy>, <color>)`

Example:

```
r <- Rectangle(30,30,20,10, MakeColor(green))
AddObject(r, win)
Draw()
```

See Also: `Circle()`, `Ellipse()`, `Square()`, `Line()`

Name/Symbol: `ReflectPoints`

Description: Takes a set of points (defined in a joined list [[x1,x2,x3,...],[y1,y2,y3,...]] and reflects them around the vertical axis x=0, returning a similar [[x],[y]] list. Identical to `ZoomPoints(pts,-1,1)`

Usage: `ReflectPoints(<points>)`

Example:

```
points <- [[1,2,3,4],[20,21,22,23]]
newpoints <- ReflectPoints(points)
```

See Also: `ZoomPoints()`, `RotatePoints`

Name/Symbol: `RemoveObject()`

Description: Removes a child widget from a parent. Useful if you are adding a local widget to a global window inside a loop. If you do not remove the object and only `Hide()` it, drawing will be sluggish. Objects that are local to a function are removed automatically when the function terminates, so you do not need to call `RemoveObject()` on them at the end of a function.

Usage: `RemoveObject( <object>, <parent>)`

Example:

See Also:

Name/Symbol: `Repeat()`

Description: Makes and returns a list by repeating `<object>` `<n>` times. Has no effect on the object. Repeat will not make new copies of the object. If you later change the object, you will change every object in the list.

Usage: `Repeat(<object>, <n>)`

Example:
```
x <- "potato"
y <- repeat(x, 10)
Print(y)
# produces ["potato","potato","potato","potato","potato",
#           "potato","potato","potato","potato","potato"]
```

See Also: `RepeatList()`

Name/Symbol: `RepeatList()`

Description: Makes a longer list by repeating a shorter list `<n>` times. Has no effect on the list itself, but changes made to objects in the new list will also affect the old list.

Usage: `RepeatList(<list>, <n>)`

Example: `RepeatList([1,2],3) # == [1,2,1,2,1,2]`

See Also: `Repeat()`, `Merge()`, `[ ]`

Name/Symbol: `Replace()`

Description: Creates a copy of a (possibly nested) list in which items matching some list are replaced for other items. `<template>` can be any data structure, and can be nested. `<replacementList>` is a list containing two-item list pairs: the to-be-replaced item and to what it should be transformed.
Note: replacement searches the entire `<replacementList>` for matches. If multiple keys are identical, the item will be replaced with the last item that matches.

Usage: `Replace(<template>,<replacementList>)`

Example:

```
x <- ["a","b","c","x"]
rep <- [["a","A"],["b","B"],["x","D"]]
Print(Replace(x,rep))
# Result:  [A, B, c, D]
```

See Also: `ReplaceChar()`

Name/Symbol: `ReplaceChar()`

Description: Substitutes `<char2>` for `<char>` in `<string>`. Useful for saving subject entry data in a file; replacing spaces with some other character.

Usage: `ReplaceChar(<string>,<char>,<char2>)`

Example:

```
x <- ["Sing a song of sixpence"]
rep <- ReplaceChar(x," ", "_")
Print(rep)
# Result:  Sing_a_song_of_sixpence
```

See Also: for list items: `Replace()`

Name/Symbol: `return`

Description: Enables a function to return a value.

Usage:

```
define funcname()
{
 return 0
}
```

Example:

See Also:

Name/Symbol: `Rotate()`

Description: Returns a list created by rotating a list by `<n>` items. The new list will begin with the `<n+1>`th item of the old list (modulo its length), and contain all of its items in order, jumping back to the beginning and ending with the `<n>`th item. Rotate(`<list>`,0) has no effect. Rotate does not modify the original list.

Usage: `Rotate(<list-of-items>, <n>)`

Example:
```
Rotate([1,11,111],1)  # == [11,111,1]
```

See Also: `Transpose()`

Name/Symbol: `RotatePoints`

Description: Takes a set of points (defined in a joined list `[[x1,x2,x3,...], [y1,y2,y3,...]]` and rotates them `<angle>` degrees around the point `[0,0]`, returning a similar `[[x],[y]]` list.

Usage:
```
ZoomPoints(<points>,<angle>)
```

Example:
```
points <- [[1,2,3,4],[20,21,22,23]]
newpoints <- RotatePoints(points,10)
```

See Also: `ZoomPoints()`, `ReflectPoints`

Name/Symbol: `Round()`

Description: Rounds `<num>` to nearest integer.

Usage: `Round(<num>)`

Example:
```
Round(33.23)  # == 33
Round(56.65)  # == 57
```

See Also: `Ceiling()`, `Floor()`, `AbsFloor()`, `ToInt()`

6.18 S

Name/Symbol: `SampleN()`

Description: Samples `<number>` items from list, returning a randomly- ordered list. Items are sampled without replacement, so once an item is chosen it will not be chosen again. If `<number>` is larger than the length of the list, the entire list is returned shuffled. It differs from `ChooseN` in that `ChooseN` returns items in the order they appeared in the originial list. It is implemented as `Shuffle(ChooseN())`.

Usage: `SampleN(<list>, <n>)`

Example:
```
SampleN([1,1,1,2,2], 5)     # Returns 5 numbers
SampleN([1,2,3,4,5,6,7], 3) # Returns 3 numbers from 1 and 7
```

See Also: `ChooseN()`, `SampleNWithReplacement()`, `Subset()`

Name/Symbol: `SampleNWithReplacement()`

Description: `SampleNWithReplacement` samples `<number>` items from `<list>`, replacing after each draw so that items can be sampled again. `<number>` can be larger than the length of the list. It has no side effects on its arguments.

Usage: `SampleNWithReplacement(<list>, <number>)`

Example:
```
x <- Sequence(1:100,1)
SampleNWithReplacement(x, 10)
# Produces 10 numbers between 1 and 100, possibly
# repeating some.
```

See Also: `SampleN()`, `ChooseN()`, `Subset()`

Name/Symbol: `SDTBeta()`

Description: `SDTBeta` computes beta, as defined by signal detection theory.

Usage: `SDTBeta(<hr>, <far>)`

Example:
```
  Print(SDTBeta(.1,.9))  #.67032
  Print(SDTBeta(.1,.5))  #.88692
  Print(SDTBeta(.5,.5))  #1
  Print(SDTBeta(.8,.9))  #0.918612
  Print(SDTBeta(.9,.95)) #0.954803
```

See Also: `SDTDPrime()`,

Name/Symbol: `SDTDPrime()`

Description: `SDTDPrime` computes d-prime, as defined by signal detection theory. This is a measure of sensitivy based jointly on hit rate and false alarm rate.

Usage: `SDTDPrime(<hr>, <far>)`

Example:

```
Print(SDTDPrime(.1,.9))  #2.56431
Print(SDTDPrime(.1,.5))  #1.28155
Print(SDTDPrime(.5,.5))  #0
Print(SDTDPrime(.8,.9))  #.43993
Print(SDTDPrime(.9,.95)) #.363302
```

See Also: `SDTBeta()`,

Name/Symbol: `SeedRNG()`

Description: Seeds the random number generator with `<num>` to reproduce a random sequence. This function can be used cleverly to create a multi-session experiment: Start by seeding the RNG with a single number for each subject; generate the stimulus sequence, then extract the appropriate stimuli for the current block. Remember to `RandomizeTimer()` afterward if necessary.

Usage: `SeedRNG(<num>)`

Example:

```
##This makes sure you get the same random order
## across sessions for individual subjects.
 SeedRNG(gSubNum)
 stimTmp <- Sequence(1:100,1)
 stim <- Shuffle(stimTmp)
 RandomizeTimer()
```

See Also: `RandomizeTimer()`

Name/Symbol: `SendData()`

Description: Sends data on network connection. Example of usage in demo/nim.pbl. You can only send text data.

Usage: `SendData(<network>,<data_as_string>)`

Example: On 'server':

```
net <- WaitForNetworkConnection("localhost",1234)
SendData(net,"Watson, come here. I need you.")
CloseNetworkConnection(net)
```

On Client:

```
  net <- ConnectToHost("localhost",1234)
  value <-  GetData(net,20)
  Print(value)
  CloseNetworkConnection(net)
##should print out "Watson, come here. I need you."
```

See Also: `ConnectToIP`, `ConnectToHost`, `WaitForNetworkConnection`, `GetData`, `ConvertIPString`, `CloseNetworkConnection`

Name/Symbol: `Sequence()`

Description: Makes a sequence of numbers from `<start>` to `<end>` at `<step>`-sized increments. If `<step>` is positive, `<end>` must be larger than `<start>`, and if `<step>` is negative, `<end>` must be smaller than `<start>`. If `<start> + n*<step>` does not exactly equal `<end>`, the last item in the sequence will be the number closest number to `<end>` in the direction of `<start>` (and thus `<step>`).

Usage: `Sequence(<start>, <end>, <step>)`

Example:
```
Sequence(0,10,3)    # == [0,3,6,9]
Sequence(0,10,1.5)  # == [0,1.5,3,4.5, 6, 7.5, 9]
Sequence(10,1,3)    # error
Sequence(10,0,-1)   # == [10,9,8,7,6,5,4,3,2,1]
```

See Also: `Repeat()`, `RepeatList()`

Name/Symbol: `SetCursorPosition()`

Description: Moves the editing cursor to a specified character position in a textbox.

Usage: `SetCursorPosition(<textbox>, <integer>)`

Example: `SetCursorPosition(tb, 23)`

See Also: `SetEditable()`, `GetCursorPosition()`, `SetText()`, `GetText()`

Name/Symbol: `SetEditable()`

Description: Sets the "editable" status of the textbox. All this really does is turns on or off the cursor; editing must be done with the (currently unsupported) device function `GetInput()`.

Usage: `SetEditable()`

Example:

```
SetEditable(tb, 0)
SetEditable(tb, 1)
```

See Also: `GetEditable()`

Name/Symbol: `SetFont()`

Description: Resets the font of a textbox or label. Change will not appear until the next `Draw()` function is called. Can be used, for example, to change the color of a label to give richer feedback about correctness on a trial (see example below). Font can alse be set by assigning to the object.font property of an object.

Usage: `SetFont(<text-widget>, <font>)`

Example:

```
fontGreen <- MakeFont("vera.ttf",1,22,MakeColor("green"),
MakeColor("black"), 1)
fontRed   <- MakeFont("vera.ttf",1,22,MakeColor("red"),
MakeColor("black"), 1)
label <- MakeLabel(fontGreen, "Correct")

#Do trial here.

if(response == 1)
{
SetText(label, "CORRECT")
SetFont(label, fontGreen)
} else {
SetText(label, "INCORRECT")
SetFont(label, "fontRed)
}
Draw()
```

See Also: `SetText()`

Name/Symbol: `SetMouseCursorPosition()`

Description: Sets the current x,y coordinates of the mouse pointer, 'warping' the mouse to that location immediately

Usage: `SetMouseCursorPosition(<x>,<y>)`

Example:

```
##Set mouse to center of screen:
SetMouseCursorPosition(gVideoWidth/2,gVideoHeight/2)
```

See Also: `ShowCursor`, `WaitForMouseButton`, `SetMouseCursorPosition`, `GetMouseCursorPosition`

Name/Symbol: `SetText()`

Description: Resets the text of a textbox or label. Change will not appear until the next `Draw()` function is called. The object.text property can also be used to change text of an object, by doing: `object.text <- "new text"`

Usage:
```
SetText(<text-widget>, <text>)
```

Example:
```
# Fixation Cross:
label <- MakeLabel(font, "+")
Draw()

SetText(label, "X")
Wait(100)
Draw()
```

See Also: `GetText()`, `SetFont()`

Name/Symbol: `Show()`

Description: Sets a widget to visible, once it has been added to a parent widget. This just changes the visibility property, it does not make the widget appear. The widget will not be displayed until the `Draw()` function is called. The .visible property of objects can also be used to hide or show the object.

Usage:
```
Show(<object>)
```

Example:
```
window <- MakeWindow()
image1  <- MakeImage("pebl.bmp")
image2  <- MakeImage("pebl.bmp")
AddObject(image1, window)
AddObject(image2, window)
Hide(image2)
Draw()
Wait(300)
Show(image2)
Draw()
```

See Also: `Hide()`

Name/Symbol: `ShowCursor()`

Description: Hides or shows the mouse cursor. Currently, the mouse is not used, but on some systems in some configurations, the mouse cursor shows up. Calling `ShowCursor(0)` will turn off the cursor, and `ShowCursor(1)` will turn it back on. Be sure to turn it on at the end of the experiment, or you may actually lose the cursor for good.

Usage: `ShowCursor(<value>)`

Example:
```
window <- MakeWindow()
ShowCursor(0)
## Do experiment here
##

## Turn mouse back on.
ShowCursor(1)
```

See Also:

Name/Symbol: `Shuffle()`

Description: Randomly shuffles a list.

Usage: `Shuffle(list)`

Example:
```
Print(Shuffle([1,2,3,4,5]))
# Results might be anything, like [5,3,2,1,4]
```

See Also: `Sort()`, `SortBy()`, `ShuffleRepeat()`, `ShuffleWithoutAdjacents()`

Name/Symbol: `ShuffleRepeat()`

Description: Randomly shuffles `<list>`, repeating `<n>` times. Shuffles each iteration of the list separately, so you are guaranteed to go through all elements of the list before you get another.

Usage: `ShuffleRepeat(<list>, <n>)`

Example:
```
Print(ShuffleRepeat([1,2,3,4,5]),3)
##  Results might be anything, like:
## [5,3,2,1,4, 3,2,5,1,4, 1,4,5,3,2]
```

See Also: `Sort()`, `SortBy()`, `ShuffleRepeat()`, `ShuffleWithoutAdjacents()`

Name/Symbol: `ShuffleWithoutAdjacents()`

Description: Randomly shuffles ¡nested-list¿, attempting to create a list where the nested elements do not appear adjacently in the new list. Returns a list that is flattened one level. It will always return a shuffled list, but it is not guaranteed to return one that has the non-adjecent structure specified, because this is sometimes impossible or very difficult to do randomly. Given small enough non-adjacent constraints with enough fillers, it should be able to find something satisfactory.

Usage: `ShuffleWithoutAdjacents(<nested-list>)`

Example:
```
Print(ShuffleWithoutAdjacents([[1,2,3], [4,5,6], [7,8,9]])
## Example Output:
## [8, 5, 2, 7, 4, 1, 6, 9, 3]
## [7, 4, 8, 1, 9, 2, 5, 3, 6]

## Non-nested items are shuffled without constraint
Print(ShuffleWithoutAdjacents([[1,2,3], 11,12,13,14,15,16]))
## output: [13, 11, 2, 14, 3, 15, 1, 16, 12]
##         [13, 12, 2, 16, 15, 11, 1, 14, 3]
##         [11, 1, 15, 2, 12, 16, 14, 13, 3]

## Sometimes the constraints cannot be satisfied.  9 will always
## appear in position 2
Print(ShuffleWithoutAdjacents([[1,2,3], 9])
## output: [3, 9, 1, 2]
##         [2, 9, 3, 1]
##         [3, 9, 2, 1]
```

See Also: `Shuffle()`, `Sort()`, `SortBy()`, `ShuffleRepeat()`, `ShuffleWithoutAdjacents()`

Name/Symbol: `Sign()`

Description: Returns +1 or -1, depending on sign of argument.

Usage: `Sign(<num>)`

Example:
```
Sign(-332.1)  # == -1
Sign(65)      # == 1
```

See Also: `Abs()`

Name/Symbol: `SignalFatalError()`

Description: Stops PEBL and prints `<message>` to stderr. Useful for type-checking in user-defined functions.

Usage: `SignalFatalError(<message>)`

Example:

```
If(not IsList(x))
{
 SignalFatalError("Tried to frobnicate a List.")
}
##Prints out error message and line/filename of function
```

See Also: `Print()`

Name/Symbol: `Sin()`

Description: Sine of `<deg>` degrees.

Usage: `Sin(<deg>)`

Example:

```
 Sin(180)
 Sin(0)
```

See Also: `Cos()`, `Tan()`, `ATan()`, `ACos()`, `ATan()`

Name/Symbol: `Sort()`

Description: Sorts a list by its values from smallest to largest.

Usage: `Sort(<list>)`

Example: `Sort([3,4,2,1,5]) # == [1,2,3,4,5]`

See Also: `SortBy()`, `Shuffle()`

Name/Symbol: `SortBy()`

Description: Sorts a list by the values in another list, in ascending order.

Usage: `SortBy(<value-list>, <key-list>)`

Example:

```
SortBy(["Bobby","Greg","Peter"], [3,1,2])
# == ["Greg","Peter","Bobby"]
```

See Also: `Shuffle()`, `Sort()`

Name/Symbol: `SplitString()`

Description: Splits a string into tokens. `<split>` must be a string. If `<split>` is not found in `<string>`, a list containing the entire string is returned; if split is equal to `""`, the each letter in the string is placed into a different item in the list. Multiple delimiters, as well as delimiters at the beginning and end of a list, will produce empty list items.

Usage: `SplitString(<string>, <split>)`

Example:
```
SplitString("Everybody Loves a Clown", " ")
# Produces ["Everybody", "Loves", "a", "Clown"]
```

See Also: `FindInString()`

Name/Symbol: `Square()`

Description: Creates a square for graphing at x,y with size `<size>`. Squares are only currently definable oriented in horizontal/vertical directions. A square must be added to a parent widget before it can be drawn; it may be added to widgets other than a base window. The properties of squares may be changed by accessing their properties directly, including the FILLED property which makes the object an outline versus a filled shape.

Usage: `Ellipse(<x>, <y>, <size>, <color>)`

Example:
```
s <- Square(30,30,20, MakeColor(green))
AddObject(s, win)
Draw()
```

See Also: `Circle()`, `Ellipse()`, `Rectangle()`, `Line()`

Name/Symbol: `Sqrt()`

Description: Square root of `<num>`.

Usage: `Sqrt(<num>)`

Example:
```
Sqrt(100)  # == 10
```

See Also:

Name/Symbol: `StDev()`

Description: Returns the standard deviation of `<list>`.

Usage: `StDev(<list>)`

Example:
```
sd <- StDev([3,5,99,12,1.3,15])
```

See Also: `Min()`, `Max()`, `Mean()`, `Median()`, `Quantile()`, `Sum()`

Name/Symbol: `Stop()`

Description: Stops a sound playing in the background from playing. Calling `Stop()` on a sound object that is not playing should have no effect, but if an object is aliased, `Stop()` will stop the file. Note that sounds play in a separate thread, so interrupting the thread has a granularity up to the duration of the thread-switching quantum on your computer; this may be tens of milliseconds.

Usage: `Stop(<sound-object>)`

Example:
```
buzz <- LoadSound("buzz.wav")
PlayBackground(buzz)
Wait(50)
Stop(buzz)
```

See Also: `PlayForeground()`, `PlayBackGround()`

Name/Symbol: `StringLength()`

Description: Determines the length of a string, in characters.

Usage: `StringLength(<string>)`

Example:
```
StringLength("absolute")     # == 8
StringLength("   spaces   ") # == 12
StringLength("")             # == 0
```

See Also: `Length()`, `SubString()`

Name/Symbol: `SubList()`

Description: Extracts a list from another list, by specifying beginning and end points of new sublist.

Usage: `SubList(<list>, <begin>, <end>)`

Example: `SubList([1,2,3,4,5,6],3,5) # == [3,4,5]`

See Also: `SubSet()`, `ExtractListItems()`

Name/Symbol: `Subset()`

Description: Extracts a subset of items from another list, returning a new list that includes items from the original list only once and in their original orders. Item indices in the second argument that do not exist in the first argument are ignored. It has no side effects on its arguments.

Usage: `Subset(<list>, <list-of-indices>)`

Example:
```
Subset([1,2,3,4,5,6],[5,3,1,1]) # == [1,3,5]
Subset([1,2,3,4,5], [23,4,2]) # == [2,4]
```

See Also: `SubList()`, `ExtractItems()`, `SampleN()`

Name/Symbol: `SubString()`

Description: Extracts a substring from a longer string.

Usage: `SubString(<string>,<position>,<length>)`

If position is larger than the length of the string, an empty string is returned. If position + length exceeds the length of the string, a string from `<position>` to the last character of the string is returned.

Example: `SubString("abcdefghijklmnop",3,5) # == "cdefg"`

See Also:

Name/Symbol: `Sum()`

Description: Returns the sum of `<list>`.

Usage: `Sum(<list>)`

Example:
```
   sum <- StDev([3,5,99,12,1.3,15])      # == 135.3
```

See Also: `Min()`, `Max()`, `Mean()`, `Median()`, `Quantile()`, `StDev()`

Name/Symbol: `SummaryStats()`

Description: Computes summary statistics for a data list, aggregated by labels in a condition list. For each condition (distinct label in the `<cond>` list), it will return a list with the following entries: `<cond> <N> <median> <mean> <sd>`

Usage: `SummaryStats(<data>,<cond>)`

Example:

```
dat <- [1.1,1.2,1.3,2.1,2.2,2.3]
cond <- [1,1,1,2,2,2]
Print(SummaryStats(dat,cond))
```

Result:

```
[[1, 3, 1.1, 1.2, 0.0816497]
, [2, 3, 2.1, 2.2, 0.0816497]
]
```

See Also: `StDev()`, `Min()`, `Max()`, `Mean()`, `Median()`, `Quantile()`, `Sum()`

6.19 T

Name/Symbol: `Tab()`

Description: Produces a tab character which can be added to a string. If displayed in a text box, it will use a 4-item tab stop.

Usage: `Tab(3)`

Example:

```
Print("Number: "  Tab(1) + number )
Print("Value: "  Tab(1) + value )
Print("Size: "  Tab(1) + size )
```

See Also: `Format()`, `CR()`

Name/Symbol: `Tan()`

Description: Tangent of `<deg>` degrees.

Usage: `Tan(<deg>)`

Example: `Tan(180)`

See Also: `Cos()`, `Sin()`, `ATan()`, `ACos()`, `ATan()`

Name/Symbol: `TimeStamp()`

Description: Returns a string containing the date-and-time, formatted according to local conventions. Should be used for documenting the time-of-day and date an experiment was run, but not for keeping track of timing accuracy. For that, use `GetTime()`.

Usage: `TimeStamp()`

Example:
```
a <- TimeStamp()
Print(a)
```

See Also: `GetTime()`

Name/Symbol: `ToInteger()`

Description: Rounds a number to an integer, changing internal representation.

Usage:
```
ToInteger(<number>)
ToInteger(<floating-point>)
ToInteger(<string-as-number>)
```

Example:
```
ToInteger(33.332)  # == 33
ToInteger("3213")  # == 3213
```

See Also: `Round()`, `Ceiling()`, `AbsCeiling()`, `Floor()`, `AbsFloor()`

Name/Symbol: `ToFloat()`

Description: Converts number to internal floating-point representation.

Usage: `ToFloat(<number>)`

Example:

See Also:

Name/Symbol: `ToNumber()`

Description: Converts a variant to a number. Most useful for character strings that are interpretable as a number, but may also work for other subtypes.

Usage:
```
ToNumber(<string)
ToNumber(<number>)
```

Example:
```
a <- ToNumber("3232")
Print(a + 1) # produces the output 3233.
```

See Also: `ToString()`, `ToFloat()`, `Round()`

Name/Symbol: `ToString()`

Description: Converts value to a string representation. Most useful for numerical values. This conversion is done automatically when strings are combined with numbers.

Usage:
```
ToString(<number>)
ToString(<string>)
```

Example:
```
a <- ToString(333.232)
Print(a + "111")
# produces the output '333.232111'.
```

See Also: `ToString()`, `+`.

Name/Symbol: `TranslateKeyCode()`

Description: Translates a code corresponding to a keyboard key into a keyboard value. This code is returned by some event/device polling functions.

Usage:

Example:

See Also:

Name/Symbol: `Transpose()`

Description: Transposes or "rotates" a list of lists. Each sublist must be of the same length.

Usage: `Transpose(<list-of-lists>)`

Example:
```
Transpose([[1,11,111],[2,22,222],[3,33,333], [4,44,444]])
# == [[1,2,3,4],[11,22,33,44],[111,222,333,444]]
```

See Also: `Rotate()`

6.20 U

Name/Symbol: `Uppercase()`

Description: Changes a string to uppercase. Useful for testing user input against a stored value, to ensure case differences are not detected.

Usage: `Uppercase(<string>)`

Example:
```
Uppercase("POtaTo")  # == "POTATO"
```

See Also: `Lowercase()`

6.21 W

Name/Symbol: `Wait()`

Description: Waits the specified number of milliseconds, then returns.

Usage: `Wait(<time>)`

Example:
```
Wait(100)
Wait(15)
```

See Also:

Name/Symbol: `WaitForAllKeysUp()`

Description: Wait until all keyboard keys are in the up position. This includes numlock, capslock, etc.

Usage:

Example:

See Also:

Name/Symbol: `WaitForAnyKeyDown()`

Description: Waits for any key to be detected in the down position. This includes numlock, capslock, etc, which can be locked in the down position even if they are not being held down. Will return immediately if a key is being held down before the function is called.

Usage:

Example:

See Also: `WaitForAnyKeyPress()`

Name/Symbol: `WaitForAnyKeyDownWithTimeout()`

Description: Waits until any key is detected in the down position, but will return after a specified number of milliseconds.

Usage: `WaitForAnyKeyDownWithTimeout(<time>)`

Example:

See Also:

Name/Symbol: `WaitForKeyDown()`

Description:

Usage:

Example:

See Also:

Name/Symbol: `WaitForKeyListDown()`

Description: Returns when any one of the keys specified in the argument is down. If a key is down when called, it will return immediately.

Usage: `WaitForKeyListDown(<list-of-keys>)`

Example: `WaitForKeyListDown(["a","z"])`

See Also:

Name/Symbol: `WaitForListKeyPressWithTimeout()`

Description: Returns when any one of the keys specified in the argument is pressed, or when the timeout has elapsed; whichever comes first. Will only return on a new keyboard/timeout events, and so a previously pressed key will not trip this function, unlike `WaitForKeyListDown()`. The `<style>` parameter is currently unused, but may be deployed in the future for differences in how or when things should be returned. Returns the value of the pressed key. If the function terminates by exceeding the `<timeout>`, it will return the string `"<unknown>"`.

Usage:
```
WaitForListKeyPressWithTimeout(<list-of-keys>,
                               <timeout>,<style>)
```

`<list-of-keys>` can include text versions of many keys. See Chapter 4, section "Keyboard Entry" for complete list of keynames.

Example:
```
x <- WaitForListKeyPressWithTimeout(["a","z"],2000,1)
if(x == "<unknown>")
{
   Print("Did Not Respond.")
}
```

See Also: `WaitForKeyListDown`, `WaitForListKeyPress`

Name/Symbol: `WaitForListKeyPress()`

Description: Returns when any one of the keys specified in the argument is pressed. Will only return on a new keyboard event, and so a previously pressed key will not trip this function, unlike `WaitForKeyListDown()` Returns a string indicating the value of the keypress.

Usage: `WaitForListKeyPress(<list-of-keys>)`

Example: `WaitForListKeyPress(["a","z"])`

See Also: `WaitForKeyListDown`, `WaitForListKeyPressWithTimeout`

Name/Symbol: `WaitForKeyPress()`

Description: Waits for a keypress event that matches the specified key. Usage of this function is preferred over `WaitForKeyDown()`, which tests the state of the key. Returns the value of the key pressed.

Usage: `WaitForKeyPress(<key>)`

Example:

See Also: `WaitForAnyKeyPress()`, `WaitForKeyRelease()`, `WaitForListKeyPress()`

Name/Symbol: `WaitForKeyUp()`

Description:

Usage:

Example:

See Also:

Name/Symbol: `WaitForMouseButton()`

Description: Waits for a mouse click event to occur. This takes no arguments, and returns a 4-tuple list, indicating:

`[xpos, ypos, button id [1-3], "<pressed>" or "<released>"]`

Usage:

```
WaitForMouseButton()
```

Example:

```
## Here is how to wait for a mouse down-click

 continue <- 1
 while(continue)
 {
     x <- WaitForMouseButton()
     if(Nth(x,4)=="<pressed>")
      {
          continue <- 0
      }
 }
 Print("Clicked")
```

See Also: `ShowCursor`, `SetMouseCursorPosition`, `GetMouseCursorPosition`

Name/Symbol: `WaitForNetworkConnection()`

Description: Listens on a port, waiting until another computer or process connects. Return a network object that can be used for communication.

Usage:

```
WaitForNetworkConnection(<port>)
```

Example: See nim.pbl for example of two-way network connection.

```
net <- WaitForNetworkConnection(1234)
dat <- GetData(net,20)
Print(dat)
CloseNetworkConnection(net)
```

See Also: `ConnectToHost`, `ConnectToIP`, `GetData`, `WaitForNetworkConnection`, `SendData`, `ConvertIPString`, `CloseNetworkConnection`

Name/Symbol: `while`

Description: 'while' is a keyword, and so is part of the syntax, not a function per se. It executes the code inside the `{}` brackets until the test inside the `()` executes as false. This can easily lead to an infinite loop if conditions are not met. Also, there is currently no break statement to allow execution to halt early. Unlike some other languages, PEBL requires that the `{}` be present.

Usage:

```
while(<test expression>)
{
 code line 1
 code line 2
}
```

Example:

```
i <- 1
while(i <= 10)
{
 Print(i)
 i <- i + 1
} # prints out the numbers 1 through 10
```

See Also: `loop()`, `{ }`

6.22 Z

Name/Symbol: `ZoomPoints`

Description: Takes a set of points (defined in a joined list [[x1,x2,x3,...],[y1,y2,y3,...]] and adjusts them in the x and y direction independently, returning a similar [[x],[y]] list.

Note: The original points should be centered at zero, because the get adjusted relative to zero, not relative to their center.

Usage:

```
ZoomPoints(points,<xzoom>,<yzoom>)
```

Example:

```
points <- [[1,2,3,4],[20,21,22,23]]
newpoints <- ZoomPoints(points,2,.5)
##Produces [[2,4,6,8],[10,11.5,11,11.5]]
```

See Also: `RotatePoints()`, `ReflectPoints`
