Amazon Reader

Amazon Reader is a program for reading a spreadsheet file formatted to hold Amazon purchase information with each row indicating an item that was purchased and the columns representing pertanent information about the the order. It allows the user to then read in from the system clipboard web pages that are clipped from the Amazon orders site and then extracts the desired information from them, and places that data into the appropriate columns of spreadsheet image, prior to saving the image back to the spreadsheet file. It also allows reading the text data from a PDF file containing the credit card charges from Amazon for the purchases and again marks the appropriate columns with the charge information to verify which items have been completed and which are still pending, along with the credit card file id that the charge information came from. This allows a semi-automated process for keeping track of the Amazon orders and their corresponding debits and credits that are charged to the credit card to verify there are no unknown charges received.

The program can be run in 3 different fashions, but for normal use of balancing the Amazon charges the GUI interface is used. This is performed by simply running the jar file with no arguments:

```
java -jar AmazonReader-1.2.jar
```

The command should be run from the parent directory of where all the spreadsheet files for Amazon are kept. This is because it also creates a Properties file that will keep track of the directories and settings selected, so that when you start the program it will automatically use those settings so you don't have to set them up again. The details of the Properties file will be explained in a later section.

The program also contains some additional test commands that run pieces of the code for test verification. These are not accessible from the GUI, but are provided by using Command-line Options. When running from the command line, the program will only perform the actions of the Options that are supplied on the command line. An example of this would be:

```
java -jar AmazonReader-1.2.jar -s testfile.ods -l 1 true -c clip1.txt -u -save
```

Note that multiple options (each option command starts with a '-' char) can be placed in a single command line. This command, for instance, would select the spreadsheet file 'testfile.ods' and load the 1st tab into memory while performing a verification that the header information contained in the file is valid. Then it would read in and parse the information from the clipboard file 'clip1.txt' (rather than reading directly from the clipboard) and update the the spreadsheet image with any new entries it gathered from it. It would then save the updated image back to the spreadsheet file.

For more extensive testing, it is rather cumbersome to write all the option commands you want to perform on a single command line. But because the command line operation executes the command options as given and then exits, there is no way to execute sequential commands where the next command depends on the previous command, unless each commad only depends on the state of the spreadsheet file and you make sure to save changes back to the file at the end of each command to make sure the next one gets those changes. This can be done, but is a slow process since it takes a several seconds for each saving and reloading of the spreadsheet file. This can be better accompplished by using the Program operation mode. In this mode, you create a script file of what you want to execute, then pass that as the only argument to the program. It will then execute all the program statements sequentially before exiting. There are also program flow statements that allow you to perform loops and conditionals, as well as variable parameters for manipulating any data captured and testing it for validation. The format for this operation is:

```
java -jar AmazonReader-1.2.jar -f myscript.scr
```

Properties File

The Properties file is a file that is used to keep track of settings that are made with the command options so that it can remember these settings from previous calls. It is created (and updated) when you run AmazonReader.jar and will be placed in the directory you execute the AmazonReader.jar file from (NOT where the jar file exists, but where you run the command from). This way, each location you execute from can have its own set of parameters that it remembers. The file is a hidden file in a hidden directory called: .amazonreader/site.properties. It contains a list of the settings to be maintained each time the program is run with each having 2 parts: the identifier tag and the value. An example file is as follows:

```
#---No Comment---
#Sun Mar 30 10:01:38 EDT 2025
DebugFileOut=debug.log
MaxLenDescription=100
MsgEnable=0x2F
PdfPath=/home/dan/Records/Finance/Credit_card_statements/2025/Chase_VISA_3996
SpreadsheetFile=NewTest.ods
SpreadsheetPath=/home/dan/Records/Finance/Amazon/Testing
SpreadsheetTab=0
TestFileOut=/home/dan/Records/Finance/Amazon/Testing/logs/test.log
TestPath=/home/dan/Records/Finance/Amazon/Testing
```

Note that it contains a comment line that shows the last date and time the file was updated. Every time one of the settings is changed by the running program. The definitions are:

`DebugFileOut`

: Defines the file name to output debug messages to in GUI mode. The file path is set to the value of `SpreadsheetPath`. If none, output will go to standard out. (not set by any command, must be set manually)

`MsgEnable`

: Selects the debug messages that are enabled (set by the `-d` option and from the selections made on the GUI panel)

`MaxLenDescription`

: Defines the maximum character length to allow for the Description field when extracting content from the clipboard to the spreadsheet. (not set by any command, must be set manually)

`PdfPath`

: Defines the path to read the PDF file from. If not defined, it will use the current directory. (set by the `-p` option and when selecting the PDF file from the GUI)

`SpreadsheetPath`

: Defines the path to read the Spreadsheet file from. If not defined, it will use the current directory. (set by the `-s` options and when selecting the Spreadsheet file from the GUI)

`SpreadsheetFile`

: Defines the name of the Spreadsheet file to read (set by the `-s` option and when selecting the Spreadsheet file from the GUI)

`SpreadsheetTab`

: Defines the starting tab number of the Spreadsheet file to load (0 for 1st tab). (set by the `-t` option and when the GUI or a command selects tab 0 or 1 when updating the spreadsheet from the Clipboard file using `-u` option or PDF file using `-p` option)

`TestPath`

: Defines the path of the TestFileOut location. (set by the `-o` option)

`TestFileOut`

: Defines the file name to output debug messages to when running from a program script. (set by the `-o` option)

Data Format Descriptions

The following indicates the format for the different categories of elements used in creating a program. The brown color represents hardcoded character values and green indicates one of the other defined data types. Braces { } can hold multiple values, each seperated by a pipe ('|') character. Optional elements are indicated by square brackets [ ]. If multiple occurrances of elements are permissable, a subscript will follow the bracket containing the number of the number of repititions allowed. Note that the braces, brackets, pipes and subscripts are not part of the format, they are just for conveying info about te format. Also, when an optional DblQuote or ParenLeft is used, the corresponding ending DblQuote or ParenRight must also be used.

WS	{ ASCII char 0x20 }$_{1\text{-}N}$	(whitespace)
PrintableChar	ASCII char 0x21 , 0x23 – 0x7E	(exclude space and double quote)
Underscore	{ _ }	
Dash	{ - }	
Comma	{ , }	
DblQuote	{ " }	
ParenLeft	{ (}	
ParenRight	{) }	
OpSign	{ + | - | * | / | % }	
CompSign	{ == | != | > | < | >= | <= }	
DecDigit	{ 0-9 }	
HexDigit	{ 0-9 | A-F | a-f }	
LowerAlpha	{ a-z }	
UpperAlpha	{ A-Z }	
NumericHex	0x {HexDigit}$_{1\text{-}8}$	
NumericDec	[Dash] {DecDigit}$_{1\text{-}10}$	
NumericLong	[Dash] {DecDigit}$_{1\text{-}19}$	
UnquotedStr	{PrintableChar}$_{1\text{-}N}$	
QuotedStr	DblQuote {PrintableChar | Space}$_{0\text{-}N}$ DblQuote	
ParamName	{UpperAlpha | LowerAlpha} {UpperAlpha | LowerAlpha | DecDigit | Underscore}$_{0\text{-}19}$	
ParamRef	$ {ParamName}	
Boolean	{TRUE | FALSE | true | false | 1 | 0}	
Integer	{NumericLong | NumericDec | NumericHex | ParamRef}	
Unsigned	{NumericDec | NumericHex | ParamRef}	
String	{UnquotedStr | QuotedStr | ParamRef}	
Array	Integer [WS] [Comma [WS] Integer]$_{0\text{-}N}$	
List	String [[WS] Comma [WhSpace] String]$_{0\text{-}N}$	
CalcSeg	[ParenLeft] {Integer} {OpSign Integer}$_{1\text{-}8}$ [ParenRight]	
Calculation	{ [ParenLeft] {CalcSeg} [ParenRight] }$_{1\text{-}8}$	
ForParams	ParamName [WS] = [WS] Integer [WS] ; [WS] CmpSign [WS] ParamValue [WS] ; [WS] Integer	
Comparison	ParamName CompSign ParamValue	
Cmd	{UpperAlpha}$_{1\text{-}N}$	(from a defined list)
Option	Dash {LowerAlpha} {LowerAlpha | DecDigit}$_{1\text{-}N}$	(from a defined list)
ParamValue	{Integer | Unsigned | String | Array | List}	
ProgCommand	{Cmd} {Comparison | ForParams | {ParamValue}$_{0\text{-}N}$ }	(depends on command)
OptCommand	[RUN] Option {ParamValue}$_{0\text{-}N}$	
Assignment	[SET] ParamName = Calculation	

Command Line Mode

Command-line Options are defined for performing some of the Amazon Logger actions. Some have associated arguments and others do not, but all are denoted by beginning with a '-' character. This is because they are also available as command line arguments as well as being used in a program script. They can be concatenated in a command line. That is, you can place more than one command line option in one line of the script. In the *Option* command list that follows, the list of arguments is indicated and each argument will begin with a character and an underscore (_). The leading character of the argument name will indicate the data type of the argument allowed:

`-d U_flags`

Sets the debug messages that are enabled.

`-s S_file`

Sets the Spreadsheet filename to load (must be an .ods file)

`-l U_numTabs B_chkHeader`

Loads the selected Spreadsheet file tabs into memory (if chkHeader true, verify the header format is valid and classify the columns for use). All operation work on this *image*.

`-t U_tab`

Sets the tab of the current Spreadsheet to use in operations

`-c S_file`

Reads the Clipboard file selection (uses a file instead of clipboard in reading Amazon orders)

`-u`

Updates the spreadsheet from the clipboards

`-p S_file`

Selects and loads the PDF file to read and mark off the items processed in the spreadsheet *image*

`-o S_file`

Sets the file to output the debug messages to (`S_file` is optional: if omitted, msgs go to stdout)

`-save`

Saves the current image info to the spreadsheet file and reloads the file back into memory

`-date L_dateFields`

Converts the list of fields into a date (assumes the date is in the future if a relative date given)

This command will return the response in the `$RESPONSE` parameter

`-datep L_dateFields`

Converts the list of fields into a date (assumes the date was in the past if a relative date given)

This command will return the response in the `$RESPONSE` parameter

`-default U_numTabs B_chkHeader`

Sets the Spreadsheet path, file and tab selections from the Properties File settings (which will be set to the last values previously used) and then loads the spreadsheet using the same arguments as the `-l` option. This keeps you from having to specify the `-s` and `-t` options every time you want to load a spreadsheet file.

`-maxcol`

Returns the number of columns defined in the current spreadsheet tab selection in the `$RESPONSE` parameter

`-maxrow`

Returns the number of columns defined in the current spreadsheet tab selection in the `$RESPONSE` parameter

`-setsize U_cols U_rows`

Resizes the spreadsheet image to the specified number of columns and rows

`-find S_orderNum`

Finds the first row containing the specified Amazon Order Number and returns it in the `$RESPONSE` parameter

`-class U_cols U_rows`

Gets the Class type of the cell in the current tab selection of the current spreadsheet image at the specified column and row location and returns it in the `$RESPONSE` parameter

`-color U_cols U_rows U_Mocolor`

Sets the background color of the specified spreadsheet image cell location to the specified color of the month (*U_MOcolor* must be a value from 1 to 12 for the month, 0 for white, or -1 for black)

`-RGB U_cols U_rows U_RGBcolor`

Sets the background color of the specified spreadsheet image cell location to the specified RGB color (6 hex digits representing 2 digits Red, 2 digits Green, 2 digits Blue)

`-HSB U_cols U_rows U_HSBcolor`

Sets the background color of the specified spreadsheet image cell location to the specified color of the month (6 hex digits representing 2 digits Hue, 2 digits Saturation, 2 digits Brightness)

`-cellget U_cols U_rows`

Returns the text data contained in the specified cell location of the current spreadsheet image and returns it in the `$RESPONSE` parameter

`-cellclr U_cols U_rows`

Clears the text contained at the specified cell location of the current spreadsheet image

`-cellput U_cols U_rows L_textFields`

Sets the text at the specified cell location of the current spreadsheet image to the fields specified

Note that all files are referenced from the current test path that is specified in the Properties file. The definitions for the data types above are:

U – unsigned 32-bit integer

I – signed 64-bit integer

B – boolean (true or false, or 1 for true and 0 for false)

S – string field (single word, no spaces)

L – field list (may have spaces, such as a series of words, but can be a single word)

Program Mode

Statements

A program consists of a series of statements, some of which are for performing specific actions, some are for control of the program flow, and others are for documenting the operation being performed.

There are 4 types of statements that are allowed:

- Comments
- Command-line Options (see previous section for details)
- Program Commands
- Parameter Assignments

Comments

Comments are any statements that begin with the '#' character (leading whitespace is allowed). These statements are ignored by the compiler.

Program Commands

`DEFINE` *`ParamName [ , ParamName ]`*$_N$

> Defines one or more parameters to be used in the script. Parameters are comma separated if more than one is defined. All parameters must be defined before they can be used. Note that the parameter names define their type, so refer to the Parameters section for more details.

`SET` *`Param = Calculation`*

> Refer to the Parameter Assignment statement for details (SET keyword is optional).

`RUN` *`Option`*

> This will run the specified Command Option (or Options, as it can have sequential options in a single command). Refer to the *Command Line Mode* section for more information of the Command Options.

`IF` *`Param CompSign Expression`*

> This is the basic conditional statement. It will verify whether the specified statement involving parameter *`Param`* is true or not. If so, the program proceeds to the next command line. If it is not true, it will proceed to the next `ELSE`, `ELSEIF` or `ENDIF` statement it finds. The *`CompSign`* value is one of { ==, !=, >=, <=, >, < }. The *`Expression`* is dependant on the type of parameter. If the parameter is numeric (Integer or Unsigned), it must be an algebraic statement that can be composed of parameters, numbers, operations and parantheses. If it is a String type, it will only take a String value (quoted or unquoted) or another String parameter. If it is a Boolean type, *`CompSign`* and *`Expression`* are omitted. Note that this last case can use a true or false to replace the Boolean parameter to allow forcing it to always or never branch (makes it easy to temporarily force a branch during testing).
>
> If the condition was met and the code following the IF statement is executed, when it gets to an `ELSE` or `ELSEIF` statement it will jump to the `ENDIF` location.

`ELSE`

This takes no arguments and performs no action. It is simply a marker for where to redirect program flow if the condition wasn't met, and an indication to jump to the `ENDIF` if the previous condition was met. This also completes the conditionals, so there can be no `ELSEIF` following this command for the current `IF` level.

`ELSEIF` *`Param CompSign Expression`*

This is the same as the `IF` statement in operation, but as an alternate condition if the previous `IF` or `ELSEIF` condition was not met.

`ENDIF`

This takes no arguments and performs no action. It is simply a marker for where to jump to when the previous conditions have concluded.

`FOR` *`Param = InitialValue ; CompSign EndValue ; StepValue`*

This allows a series of commands to be repeated for a specified number of times. It has 3 argument sections, each seperated by a semicolon. The format of the arguments is given by the `ForParams` entry in the *Data Format Descriptions* section. The first is the loop parameter name followed by and equals sign and the *`InitialValue`* to initialize it to for its first pass through the loop. This can be either a hard-coded value or a parameter reference. The second section specifies the *`EndValue`* for the loop parameter – in other words, when this condition is met the loop will be exited. The *`CompSign`* value is one of { ==, !=, >=, <=, >, < }. The third and final section specifies *`StepValue`*, the increment size of the loop parameter (how much to add to it after each loop) and is optional. This value can be a positive or negative value and can also be a parameter reference. If omitted, the step size is an increase of 1. The loop name follows the same rules as for the String parameter, but must be different than any parameter name. Loops can also be nested, but the loop parameter name must be different for each loop. The loop parameter name can be reused once outside its current loop. Note that the loop will not run at all if the *`InitialValue`* exceeds the conditions of the EndValue.

`BREAK`

This is used within the most recent `FOR` loop and will cause the program to jump to the next `ENDFOR` statement.

`CONTINUE`

This is used within the most recent `FOR` loop and will cause the program to jump to the following `NEXT` command, which will increase the loop parameter by the *`StepValue`* and then test to see if the *`EndValue`* condition has been met. If so, it will jump to the next `ENDFOR` statement, otherwise it will jump back to the beginning of the loop.

`NEXT`

This is used within the most recent `FOR` loop and will cause the program to increase the loop parameter by the *`StepValue`* and then test to see if the *`EndValue`* condition has been met. If so, it will jump to the next `ENDFOR` statement, otherwise it will jump back to the beginning of the loop.

`ENDFOR`

This takes no arguments and performs no action. It is simply a marker for where to jump to when the loop has completed.

Parameter Assignments

Sets a parameter to the specified value. The value can be a concrete entry (such as 125 or "hello") or can be a parameter or a formula such as $RESULT * 4. Note that parameters in the formula must all be preceeded with a '$' and care must be taken to only assign numeric entries to the numeric parameters (I_, U_ and A_). This is equivalent to the Parameter Assignment statement (SET keyword is optional).

Parameters

Parameters are a feature that allows data to be collected, manipulated, reported and used to allow commands to be issued with dynamically changing values. The format of all parameter names must begin with an alpha character (A-Z or a-z) and be composed of only alphanumeric characters and the underscore character. All parameters are case sensitive (Hench Param1 and param1 are distinctly different names). When defining a parameter, you simply use the name of the parameter to defined followed by an '=' sign and then the value to assign to it. When using a parameter in a command (or as the right-hand field of an assignment) you must preceed the name with a '$' character to indicate you are specifying a parameter and not a String value.

There are 5 types of parameters that can be defined:

- *Integers* – they are defined by the name starting with "I_"
- *Booleans* – they are defined by the name starting with "B_"
- *Strings* – they are defined by default, the name starts with anything other than I_, B_, A_ or L_.
- *Arrays* – they are an array of Integer types that are defined by the name starting with "A_"
- *Lists* – they are an array of String types that are defined by the name starting with "L_"

Integers

Integers consist of either an optional sign followed by numeric digits, or "x" or "0x" followed by hexadecimal digits. If hexadecimal digits are represented, the value is taken as an unsigned 32-bit integer value having a range of 0x00000000 to 0xFFFFFFFF. *String* parameters can be assigned to an *Integer* parameter only if they follow the previous rule mentioned on the requirements for an *Integer* value. *Boolean* parameters can be assigned to an *Integer* parameter where a TRUE value will be converted to a 1 and FALSE to a 0. The following operations are allowed for *Integers*: { +, -, *, /, % } and for unsigned 32-bit values the following additional operators: { SHIFTL, SHIFTR, ROL, ROR, NOT, AND, OR, XOR }.

Operations

The operations for *Integers* work as follows:

+	Addition
-	Subtraction
*	Multiplication
/	Integer division (truncates result)
%	Modulus (get remainder of integer division

The operators can either be used as `Parameter = Value1 OP Value2` , or as `Parameter OP= Value1` , where `Parameter` is the parameter being modified, `OP` is the specified operation and `Value` can either be an *Integer* or a Parameter having an *Integer* value.

The additional 32-bit unsigned *Integer* values can also use the following operators (Note: the value must be an unsigned 32-bit value and the result of the operation will always keep the result as an unsigned 32-bit value):

ROL ***bits***	Rotate bits left by specified amount of ***bits*** (for N = 1 to 31, bit_N gets bit_{N-1} and bit_0 gets bit_{31} value)
ROR ***bits***	Rotate bits right by specified amount of ***bits*** (for N = 0 to 30, bit_N gets bit_{N+1} and bit_{31} gets bit_0 value)
NOT	Invert all 1's and 0's
AND ***value***	bitwise AND with ***value***
OR ***value***	bitwise OR with ***value***
XOR ***value***	bitwise exclusive-OR with ***value***

Examples

```
I_Value = 127 * $I_Mult
I_FlagBits XOR= 0x007F
```

Booleans

Booleans consist of the values *TRUE* or *FALSE* (or *true* / *false*). If an *Integer* value is assigned to it, a 0 will be converted to FALSE and a 1 will be converted to TRUE, with any other value causing an error. If a *String* value is assigned to it, it must be composed of either the values TRUE or FALSE (case insensitive) or a numeric value that will again be interpreted as FALSE for 0 and TRUE otherwise. No operations are valid for *Booleans*, but it can take a *String* or *Integer* comparison value that returns a *Boolean* result.

Examples

```
B_Value = TRUE
B_Value = $I_Value > 75
```

Strings

Strings consist of any ASCII printable character (values 0x20 – 0x7E), which may be enclosed in double quotes. If the String is to consist of multiple words, that is, some spaces will be included in it, you MUST enclose the String with quotes. If it is a single word, the quotes are optional. *Integers* and *Booleans* can be assigned to a *String* parameter (the Boolean value will be converted to "TRUE" or "FALSE"). Operations that are allowed for *Strings*: { +, CONCAT, TRUNCL, TRUNCR, LENGTH, UPPER, LOWER, SUB }.

Operations

The operations for *Strings* work as follows:

+ ***StrVal***	Concatenate StrVal to the right of the current *String* value
.CONCAT ***StrVal***	Concatenate StrVal to the right of the current *String* value
.TRUNCL ***num***	Truncate num characters from the beginning of the *String* (remove chars to the left)
.TRUNCR ***num***	Truncate to the first num characters of the *String* (eliminate all chars to the right)
.UPPER	Converts the *String* to all uppercase characters
.LOWER	Converts the *String* to all lowercase characters
.LENGTH	returns *Integer* length of the *String*

Examples

```
MyString = First                        MyString is now: First
MyString = MyString.CONCAT Second       MyString is now: FirstSecond
MyString = MyString.TRUNCL 2            MyString is now: rstSecond
MyString = MyString.UPPER               MyString is now: RSTSECOND
I_Length = MyString.LENGTH              I_Lemgth is now: 9
```

Arrays

Arrays consist of a comma or space-separated series of 0 or more *Integer* values. You can access (read or modify) any entry in the *Array* using the square brackets with the numeric index (starting at 0), such as **`I_Value = $A_Table[6]`** to get the 7th value in the Array and assign it to the parameter I_Value. If an assignment is done to an *Integer* or the use in a command requires a *Integer* instead of an *Array* and the parameter is an *Array* type, only the 1st element will be used from the *Array* (element [0]). The valid operations for *Arrays* are: { ADD, REMOVE, PUSH, POP, SIZE, ISEMPTY, LIST }.

Operations

The operations for *Arrays* work as follows:

.ADD ***IntVal***	Add **IntVal** to the end of the current *Array* value (can optionally have space-seperated additional values)
.REMOVE	Remove the last entry from the current *Array* value (value will be placed in **`$RESPONSE`**)
.PUSH ***IntVal***	Add **IntVal** to the begining of the current *Array* value
.POP	Remove the first entry from the current *Array* value (value will be placed in **`$RESPONSE`**)
.INSERT ***ix IntVal***	Add **IntVal** to index **ix** of the current *Array* value

.SIZE	returns *Integer* number of entries in the *Array*
.ISEMPTY	returns *Boolean* **TRUE** if the the *Array* is empty, **FALSE** otherwise

Examples

```
A_Table1 = 22, 25, 27, 29                    A_Table1 is now: 22 25 27 29
A_Table2 = $I_Values 16                      A_Table2 is now: 10 16
A_Table1.REMOVE                              A_Table1 is now: 22 25 27
A_Table2.PUSH $RESPONSE                      A_Table2 is now: 29 10 16
```

Lists

Lists consist of a space-separated series of 0 or more *String* formatted words. You can access (read or modify) any entry in the *List* using the square brackets with the numeric index (starting at 0), such as **StrValue = $L_Table[3]** to get the 4h value in the *List* and assign it to the parameter StrValue. The individual *String* element returned from a [x] nomenclature follows the same rules as any other *String*. If an assignment is done to a *String* or the use in a command requires a *String* instead of a *List* and the parameter is a *List* type, only the 1st element will be used from the *List* (element [0]). If the command requires or allows a *List* type, using the *List* name will supply the entire list of entries. The valid operations for *Arrays* are: { ADD, REMOVE, PUSH, POP, SIZE, ISEMPTY, LIST }.

Arrays and *Lists* can also be assigned to a *Boolean* parameter using the element selection [x] to identify which element is being referred to (or will default to the 1st element).

When parameters are referenced in an assignment to a parameter or in a command as a replacement for a value, they must be preceeded by the '$' character to indicate this. If the parameter is not found and the assignment is to a *String* or *List* parameter, the value will be accepted at face value. That is, if the command **StrParam = $NotFound** is used and **NotFound** is not a defined parameter, the value of StrParam will be "$NotFound".

Operations

The operations for *Lists* work as follows:

.ADD ***StrVal***	Add **StrVal** to the end of the current *List* value
.REMOVE	Remove the last entry from the current *List* value (value will be placed in **$RESPONSE**)
.PUSH ***StrVal***	Add **StrVal** to the begining of the current *List* value
.POP	Remove the first entry from the current *List* value (value will be placed in **$RESPONSE**)
.INSERT ***ix StrVal***	Add **StrVal** to index **ix** of the current *List* value
.SIZE	returns *Integer* number of entries in the *List*
.ISEMPTY	returns *Boolean* **TRUE** if the the *List* is empty, **FALSE** otherwise

Examples

```
L_Table1 = This is a List sample             L_Table1 is now: This is a List sample
L_Table2 = Silly                             L_Table2 is now: Silly
L_Table1.INSERT 3 $L_Table2                  L_Table1 is now: This is a Silly sample
L_Table2 = $L_Table2.POP                     L_Table2 is now:
B_Status = $L_Table2.ISEMPTY                 B_Status is now: TRUE
```