Quick Reference

Links

- Using the Functions Library
- Functions by Category
- Function Summary
- Function Definitions
- Glossary

Functions Library Version: 1-0
ETAC Programming Language Version: 1-1
ETAC Interpreter Version: 3-0-6-ena

Functions Library

Inclusion File: etacFunctions.etac
Data Object Name: etacFunctions
Number of Functions: 60
Release Date: 1 August 2020

Functions ETAC Script Library

1 August 2020

Other Related ETAC Documents

ETAC_Preliminaries.pdf	Preliminaries before using ETAC
ETACOverview.pdf	An Overview of ETAC
ETACProgLang(Official).pdf	The Official ETAC Programming Language
RunETAC.chm	Run ETAC Scripts Help
ETACWithCPP.pdf	ETAC: Interacting with C++
ETACCompiler.pdf	The ETAC Compiler
ETACCompiler.chm	ETAC Compiler Help
ETACErrorCodes.pdf	ETAC Compilation and Run-time Error Codes

Legal Information

Contents

Document Conventions

The following symbolic conventions are used in this document.

Symbol	Meaning
‹x›	separates x as a unit of information from the surrounding text.
$x\cdots$	middle ellipsis means zero, one, or more of the same kind as x.
[x]	means that x optional.
{x}	means that x is the default value.
(x)	groups x as a unit.
$x \mid y$	means that only x or y applies, but not both (could have more than two options).
...	ellipsis represents omitted text (as usual).
WS	represents a whitespace character (9_{16} to D_{16}, or 20_{16}).
SP	represents a space character (20_{16}).
CR	represents a carriage return character (D_{16}).
LF	represents a linefeed character (A_{16}).
EL	represents the character or characters indicating the end of a text line.
n_{16}	represents a number in base 16 (hexadecimal).
U+x	represents a Unicode code point where x is in hexadecimal notation.
text	maroon coloured italic text is a link to the text's definition.
text	underlined green text is a link into the document.
text	bold green text is a link into the document.
♦	indicates the end of a block of text.

Functions ETAC Script Library

This document defines the functions in the Functions Library for version **1-1** of the **ETAC™ Programming Language** implemented in program RunETAC.exe version **3-0-6-ena**. The Functions Library described in this document is version **1-0**.

(Australian English)

1. Introduction

ETAC™ (pronounced: E-tack) is a syntactically simple but extremely versatile dictionary and stack based interpreted script programming language. The ETAC programming language is not based on any other programming language. Familiarity with the ETAC programming language is required to fully understand this document. See the document ETACOverview.pdf for an overview of the language, and also the document ETACProgLang(Official).pdf for the official definition of the language.

Since ETAC is a script programming language, libraries of *ETAC functions* can be written in *ETAC text script* for use in ETAC programs. Such libraries are called *ETAC script libraries* (abbreviated as "ESL"). This document contains the definitions and use of one such library for general purpose usage. The library is referred to as the "Functions Library".

The Functions Library is an *ETAC script library* containing *ETAC functions* defined in *ETAC text script*. Desired functions from the Functions Library can be included into any *ETAC text script* via the etacFunctions.etac inclusion file. The functions in the Functions Library are not defined for any particular purpose, and are for general use.

2. Features of an ETAC Script Library

An *ETAC script library* (ESL) is an *ETAC text script* file containing *ETAC function* definitions. The *ETAC functions* are constructed in such a way that the same function is allocated only once within a *main ETAC session*, even when the same ESL file is included more than once.

An ETAC programmer can arrange for an ESL so that only the desired *ETAC functions*, and their dependencies, are allocated within a *main ETAC session*. This feature is useful when only a few functions of an ESL containing a large number of *ETAC function* definitions are required in an ETAC program. By default, all of the *ETAC functions* in an ESL are included in an ETAC program.

The included functions of an ESL are typically allocated within a named *data object* having a name based on the name of the ESL.

3. Using an ETAC Script Library

An *ETAC script library* (ESL) exists in an *ETAC text script* file, and can be included in any *ETAC text script* as an inclusion file (using the ‹::include› pre-processor inclusion command). An ESL is typically included at the top of the *ETAC text script*. The processing of the ESL inclusion file causes a new named *data object* to be created containing the desired *ETAC function* allocations; the *data object* name is based on the name of the ESL. For example, the *data object* name into which the desired functions of the Functions Library are included is called `etacFunctions`.

Each *ETAC function* within an ESL is identified by a pre-processor definition name beginning with the initials of the ESL name followed by the name of the particular function. For example, if an ESL function name is `xxxIndentLines`, then the corresponding pre-processor definition name would be `@XXX_INDENT_LINES`, where *xxx* and *XXX* are the initials of the ESL name (for the Functions Library, *XXX* is `F`, so the function name identifiers are `fIndentLines` and `@F_INDENT_LINES`, respectively).

The pre-processor definition name of each *ETAC function* within an ESL is called the *ESL function identifier* (for example, `@F_INDENT_LINES` is an *ESL function identifier*). To explicitly include a particular ESL function within some *ETAC text script*, the corresponding ESL function identifier must be defined before the ESL inclusion file is included. For example, to include the `fIndentLines` function of the Functions Library into *ETAC text script*, ‹`::define @F_INDENT_LINES`› must be present before ‹`::include "etacFunctions.etac"`›. If the included function calls other ESL functions, then those other ESL functions are automatically included as well.

Including an ESL inclusion file without specifying the definition of any *ESL function identifier* of that ESL results in all of the functions in that ESL being included. For example, specifying only ‹`::include "etacFunctions.etac"`› without defining any of the *ESL function identifiers* of the Functions Library beforehand (in the same *ETAC text script*) results in all of the *ETAC functions* within the Functions Library to be included into the *data object* `etacFunctions`.

If an *ESL function identifier* is defined more than once for the same ESL inclusion file, then the corresponding *ETAC function* is allocated only once within the *main ETAC session*, even when the same ESL inclusion file is included more than once.

In summary, to use an *ETAC script library*, the following are required:

- The file name of the ESL, which must be included before any of the ESL functions are used.
- A list of *ESL function identifiers* (corresponding to the desired ESL functions), which must be defined before the inclusion of the ESL file. The identifiers are obtained from the ESL source file or documentation.
- The name of the ESL *data object* (obtained from the ESL source file or documentation), which is automatically created by the inclusion of the ESL.
- A variable used to access the ESL *data object*. The desired ESL functions are accessed via that variable, which is assigned from the return value of the global **`@NewData`** *ETAC function*.

The following example is an illustration of how to use the Functions Library in a main ETAC program.

Using the Functions Library

```
[*::10ETAC-SOURCE-V1::*]
[* ETAC main program file illustrating how to use the Functions Library. *]
::include "TACGlobal.PTAC" [* Required by the Functions Library. *]

...

::define @F_INDENT_LINES [* Allows allocation of fIndentLines function. *]
::define @F_STR_TO_LINES [* Allows allocation of fStrToLines function. *]
::define @F_CREATE_FILE [* Allows allocation of fCreateFile function. *]
::include "etacFunctions.etac" [* Functions Library inclusion file. *]
[* etacFunctions.etac creates the etacFunctions data object containing the three functions. *]

...

start_local; [* Create a local dictionary for the main program. *]
[* Other program variables allocated here. *]

   Fnts :- @NewData("etacFunctions"); [* Allocate Fnts variable for accessing ESL functions. *]

   [* PROGRAM *]
   ...

   Seq := Fnts.fStrToLines(Str "[{\r\n}\r\n]"); [* Call to ESL function. *]

   ...

end_local;
```

In the example above, three *ETAC functions*, `fIndentLines`, `fStrToLines`, and `fCreateFile`, defined in the Functions Library are to be used in the program (the three functions are for illustrative purposes only). The corresponding *ESL function identifiers* of those functions are defined (via `::define`) so that the three functions are automatically allocated in the `etacFunctions` *data object* when the Functions Library inclusion file, `etacFunctions.etac`, is included in the program. The three *ESL function identifiers* must be defined before the inclusion of the Functions Library inclusion file. If no *ESL function identifiers* were defined, then all the *ETAC functions* of the Functions Library would have been allocated in the `etacFunctions` *data object*, even though only some of those functions are used in the program. Those three functions will not be allocated again within the `etacFunctions` *data object*, even if they were to be specified for inclusion in another *ETAC session*.

Somewhere in the program, after the inclusion of the Functions Library, a replicate of the `etacFunctions` *data object* is created by the global **@NewData** function, and allocated to a variable, `Fnts`, to access the *ETAC functions* within the *data object*. The `etacFunctions` *data object* is replicated (via **@NewData**) rather than used directly (via **@Data**) so as to keep the original *ETAC functions* within the *data object* unchanged; *ETAC functions* within an ESL may contain initialised persistent local data which needs to remain as originally initialised. Using an ESL *data object* directly may cause undesirable changes in any persistent local data within that *data object*.

4. Functions ETAC Script Library Reference

This section contains information about the Functions ETAC Script Library (referred to as the “Functions Library”). The Functions Library exists in the etacFunctions.etac inclusion file. When that file is included (via `::include`) in *ETAC text script*, a *data object* named `etacFunctions` is automatically created containing the desired functions. Those functions can then be assessed via a replicate of that *data object*.

4.1 Functions by Category

The following is a list of the Functions Library functions organised by category.

Disk File

fCreateFile ♦ **fIsFileWritable** ♦ **fPathExists** ♦ **fWriteFile**

File Data and Memory

fCreateFile ♦ **fGetMemSize** ♦ **fLinesToMem** ♦ **fMemToHexChars** ♦ **fReadTextLines** ♦ **fWriteFile**

File Path

fCvtRelativePath ♦ **fGetWindowsDir** ♦ **fIsOnlyDirPath** ♦ **fIsOnlyFileName** ♦ **fIsRelativePath**

Number

fAlignVal ♦ **fCeil** ♦ **fFloor** ♦ **fFromWinCC** ♦ **fHexToInt** ♦ **fToBinStr** ♦ **fToBoolStr** ♦ **fToHexStr** ♦ **fToWinCC**

String

fBlotStrChars ♦ **fCaptureComments** ♦ **fCaptureQuotes** ♦ **fExtractInnerStr** ♦ **fFormatStr** ♦ **fGetStrU** ♦ **fIsStrDblQuoted** ♦ **fIsStrDec** ♦ **fIsStrInt** ♦ **fIsStrNegInt** ♦ **fIsStrPosInt** ♦ **fIsStrZeroInt** ♦ **fMatchString** ♦ **fMid** ♦ **fParseString** ♦ **fPutStrU** ♦ **fRemQuotes** ♦ **fRepeatStr** ♦ **fReplSubStr** ♦ **fStrToLines** ♦ **fTrimQuotes** ♦ **fTrimStrWS**

String Sequence

fDelDuplStrs ♦ **fIndentLines** ♦ **fLinesToMem** ♦ **fLinesToStr** ♦ **fQuickSort** ♦ **fReadTextLines** ♦ **fSortSeq** ♦ **fStrInSeq** ♦ **fStrToLines**

Unicode

fCharToCP ♦ **fCPToChar** ♦ **fGetStrU** ♦ **fPutStrU**

Other

fDateTimeFormatted ♦ fExecETACStr ♦ fGetKWArgs ♦ fGetKWSyntax ♦ fLambda ♦ fLambdaApp ♦ fShowBusy ♦ fRunETACFile

4.2 Function Summary

The table below contains an alphabetical list of the Functions Library functions.

Function Summary for the Functions Library

Function	Description
fAlignVal	Rounds up an integer value to the next specified byte alignment.
fBlotStrChars	Replaces a length of characters in a string with a repeated character.
fCaptureComments	Returns a sequence identifying the locations and lengths of ETAC comments within a string.
fCaptureQuotes	Returns a sequence identifying the locations and lengths of quoted substrings within a string.
fCeil	Returns the ceiling of a decimal number.
fCharToCP	Converts the first character of a string to a Unicode code point.
fCPToChar	Converts a Unicode code point to a character.
fCreateFile	Creates a new empty file if it does not exist.
fCvtRelativePath	Returns the full path of a file path, which may be relative to a specified directory path.
fDateTimeFormatted	Returns a formatted date and time string of the current date and time.
fDelDuplStrs	Deletes duplicate elements of a string sequence.
fExecETACStr	Executes a string containing ETAC text.
fExtractInnerStr	Extracts the substring from a string enclosed within brackets.
fFloor	Returns the floor of a decimal number.
fFormatStr	Replaces certain symbols within a format string.
fFromWinCC	Converts a Windows-1252 character code to a string.
fGetKWArgs	Processes keywords and their arguments.
fGetKWSyntax	Gets the keyword-arguments syntax of a keyword template.
fGetMemSize	Returns the byte size of the usable data in a memory object.
fGetStrU	Returns the middle part (specified as *u-chars*) of a string.
fGetWindowsDir	Get the full path of the system `Windows` directory.
fHexToInt	Converts the characters of a hexadecimal string to an integer.
fIndentLines	Indents all text lines in a string sequence.
fIsFileWritable	Checks that a file is writeable if it exists.
fIsOnlyDirPath	Determines whether a path specification is a directory path only.
fIsOnlyFileName	Determines whether a file path specification contains only a file name (and extension).
fIsRelativePath	Determines whether a file path specification is a relative path.
fIsStrDblQuoted	Determines whether a string is delimited by double-quote characters.
fIsStrDec	Determines whether a string is in the form of decimal number.
fIsStrInt	Determines whether a string is in the form of an integer.
fIsStrNegInt	Determines whether a string is in the form of a negative integer.
fIsStrPosInt	Determines whether a string is in the form of a positive integer.

fIsStrZeroInt	Determines whether a string is in the form of a zero integer.
fLambda	General lambda abstraction function creator for any predefined function or procedure.
fLambdaApp	General lambda application function creator for any predefined function or procedure.
fLinesToMem	Converts a string sequence to a memory object with EOL characters.
fLinesToStr	Converts a string sequence to a string with EOL characters.
fMatchString	Matches a string based on a *pattern string*.
fMemToHexChars	Converts a portion of a memory object into a hexadecimal string.
fMid	Returns the middle part of a string.
fParseString	Parses a string based on a *pattern string* and possibly sub-patterns.
fPathExists	Determines whether a specified type of disk entity exists for a path specification.
fPutStrU	Replaces a substring (specified as *u-chars*) in a string.
fQuickSort	Sorts a sequence in place using the quick sort algorithm.
fReadTextLines	Reads the text lines of a text file into a string sequence.
fRemQuotes	Trims a string by removing leading and trailing single or double quotes and then spaces.
fRepeatStr	Creates a string from repeats of a given string.
fReplSubStr	Replaces all substrings of a string matching a *pattern string*.
fRunETACFile	Runs an ETAC (or TAC) file as it would be run from `RunETAC.exe`.
fShowBusy	Shows or hides a busy message.
fSortSeq	Sorts a sequence in place using the insertion sort algorithm.
fStrInSeq	Returns the index of a substring existing in a string sequence.
fStrToLines	Converts a string containing EOLs to a sequence of strings.
fToBinStr	Converts an integer to a binary string.
fToBoolStr	Converts a boolean value to a string representing that value.
fToHexStr	Converts an integer to a hexadecimal string.
fToWinCC	Converts a character to its Windows-1252 character code.
fTrimQuotes	Removes single and double quotes from only the ends of a string.
fTrimStrWS	Trims a string by removing leading and trailing whitespaces.
fWriteFile	Writes data to the specified file possibly with backup.

"EOL" stands for "end-of-line".

4.3 Function Definitions

The following boxes contain a description of all the Functions Library functions. The functions need to be called in the context of the `etacFunctions` *data object*. The corresponding *ESL function identifier* is shown at the top right of each box.

fAlignVal

fAlignVal *val align* → *res*	@F_ALIGN_VAL
val	A non-negative integer stack object.
align	A positive integer stack object.
res	A non-negative integer stack object.

Details

Rounds up an integer value (*val*) to the next specified byte alignment (*align*) returning the result (*res*). If *val* is already aligned as specified then *res* will be identical to *val*.

align must be a positive power of 2 (ie: 2, 4, 8, 16, ⋯), otherwise the consequence is undefined.

Examples

The following illustrations show how the **fAlignVal** function can be used.

```
(1) fAlignVal(22 4);
(2) Val := fAlignVal(22 2);
```

The call in example **(1)** returns `24` because the argument, `22`, is not aligned on the next `4` byte boundary, which is `24`.

In example **(2)**, `Val` will be `22` because that value is already aligned on a `2` byte boundary. ♦

fBlotStrChars

fBlotStrChars *offset len char str* → *out-str*	@F_BLOT_STR_CHARS
offset	A non-negative integer stack object.
len	A non-negative integer stack object.
char	A string stack object.
str	A string stack object.
out-str	A string stack object.

Details

Replaces a *w-char* length (*len*) of *w-char* characters in a string (*str*) at a *w-char* offset (*offset*) with a string of a repeated UCS-2 character (*char*) of the same length.

offset is a zero-based *w-char* character offset into *str*. If *offset* indicates a character beyond the last *w-char* character of *str*, then no action occurs and *out-str* will be identical to *str*. The character at *offset* must represent a Unicode scalar value.

len is the maximum number of *w-char* characters to be replaced in *str* beginning at *offset*. The substring in *str* to be replaced must be a well-formed Unicode substring. If *len* exceeds the remaining characters of *str*, then only the remaining characters are replaced. *len* is also the number of repeated first character of *char* to replace the substring of *str*.

Only the first character of *char* is used, which must be a UCS-2 (BMP Unicode scalar value) character. That character is repeated *len* times and replaces up to *len* *w-char* characters in *str*.

Examples

The following illustrations show how the **fBlotStrChars** function can be used.

```
(1) fBlotStrChars(5 1 " " "hello-ha");
(2) fBlotStrChars(6 4 "*%" "hello-ha");
```

```
(3) fBlotStrChars(8 4 "*%" "hello-ha");
(4) fBlotStrChars(6 2 "*" "thumbs\#1F44D#up");
(5) fBlotStrChars(1 3 "" "hello-ha");
```

Example **(1)** returns the string ‹hello ha›.

Example **(2)** returns the string ‹hello-****›.

Example **(3)** returns the string ‹hello-ha› because *offset* is beyond the last character of *str*.

Example **(4)** returns the string ‹thumbs**up›. Note that the Unicode supplementary plane code point U+1F44D (Thumbs Up Sign) is two *w-char* characters wide. Because *len* is the *w-char* character length, both *w-chars* (surrogate pairs) of the character at *offset* 6 are replaced, resulting in two asterisks rather than one. Also note that if *len* were 1, or *offset* were 7, then an *error event* would have occurred because the substring being replaced would not have been a well-formed Unicode string.

Example **(5)** returns the string ‹hello-ha› because the specified substring is not replaced since *char* does not contain a first character. ♦

fCaptureComments

fCaptureComments *str* → *seq*	@F_CAPTURE_COMMENTS
str	A string stack object.
seq	A sequence of sequences.

Details

Returns (*seq*) a sequence of *w-char* character offset\length pairs (in a sequence) identifying the locations and lengths of ETAC comments within a string (*str*). The comments can be nested.

The comment delimiters in *str* need to be properly matched as single-asterisk delimiters to be recognised correctly (each ‹[*› must match with ‹*]›).

The returned sequence, *seq*, will contain zero or more two-element sequences. Each two-element sequence identifies the location of a comment within *str*. The first element of a two-element sequence will be a *w-char* character offset integer, and the second element will be a *w-char* character length integer. If there are no comments in *str*, then an empty sequence will be returned. The returned sequence can be used directly with the **get_str** and **put_str** ETAC commands.

Examples

The following illustrations show how the **fCaptureComments** function can be used.

```
(1) fCaptureComments('This string [*with [*comments*] here*] and [*here*]');
(2) fCaptureComments('This string [*with comments**] here*] and [*here*]');
```

Example **(1)** returns the sequence [[12, 26], [43, 8]], indicating the locations (offset\length pairs) of the (highlighted) comments within the string argument.

Example **(2)** returns the sequence [[12, 18]], indicating the (highlighted) comment ‹[*with comments**]›, even though the string argument contains the two comments ‹[*with comments**] here*]› and ‹[*here*]›. The reason that the comments were not properly captured is that **fCaptureComments** only recognises matched and nested ‹[*› and ‹*]› as comment delimiters. ♦

fCaptureQuotes

fCaptureQuotes *str* → *seq*	@F_CAPTURE_QUOTES
str	A string stack object.
seq	A sequence.

Details

Returns (*seq*) a sequence of *w-char* character offset\length pairs (in a sequence) identifying the locations and lengths of single-quoted (U+0027) or double-quoted (U+0022) substrings within a string (*str*). Backslash escaped quotes are ignored within the quoted substrings.

The string delimiters in *str* need to be properly matched to be recognised correctly (each ‹"› must match with the next un-escaped ‹"›, and each ‹'› must match with the next un-escaped ‹'›).

The returned sequence, *seq*, will contain zero or more two-element sequences. Each two-element sequence identifies the location of a single or double-quoted substring within *str*. The first element of a two-element sequence will be a *w-char* character offset integer, and the second element will be a *w-char* character length integer. If there are no single or double-quoted substrings in *str*, then an empty sequence will be returned. The returned sequence can be used directly with the **get_str** and **put_str** ETAC commands.

Example

The following illustration shows how the **fCaptureQuotes** function can be used.

```
(1) fCaptureQuotes('This string "with \"quotes\" here" and '¯here'¯');
```

Example **(1)** returns the following sequence `[[12, 22], [39, 6]]`, indicating the (highlighted) locations (offset\length pairs) of the quoted substrings within the string argument. Note that ‹'¯› represents a single quote character within a single quoted ETAC string. ♦

fCeil

fCeil *num* → *ceil*	@F_CEIL
num	A decimal or integer stack object.
ceil	A decimal or integer stack object.

Details

Returns (*ceil*) the ceiling of a decimal or integral number (*num*). The ceiling of a number is the smallest integer greater than or equal to that number.

If *num* is an integer then *ceil* will also be an integer equal to *num*. If *num* is a decimal then *ceil* will be an integer if it is within the range of an integer (–2,147,483,648 to 2,147,483,647), otherwise it will be an integral decimal. ♦

fCharToCP

fCharToCP *char* → *cp*	@F_CHAR_TO_CP
char	A string stack object.
cp	An integer stack object.

Details

Returns the first *u-char* character of a string (*char*) as a Unicode scalar value. If *char* is an empty string, then *cp* will be zero. ♦

fCPToChar

fCPToChar *cp* → *char*	@F_CP_TO_CHAR
cp	An integer stack object.
char	A string stack object.

Details

Converts a Unicode code point (*cp*) to a *u-char* character (*char*). If *cp* is zero or not a Unicode scalar value (a Unicode surrogate code point is not a scalar value), *char* will be an empty string. ♦

fCreateFile

fCreateFile *file-path*	@F_CREATE_FILE
file-path	A string stack object.

Details

Creates a new empty disk file as specified (*file-path*) if it does not exist on disk. If the specified file already exists on disk, this function has no effect. An *error event* will occur if the file cannot be created. ♦

fCvtRelativePath

fCvtRelativePath *dir-path* *file-path* → *path-str*	@F_CVT_RELATIVE_PATH
dir-path	A string stack object.
file-path	A string stack object.
path-str	A string stack object.

Details

Returns the full file path specification (*path-str*) of a specified file path (*file-path*), which may be relative to a specified directory path (*dir-path*). If *file-path* is a relative path then the returned path specification is relative to *dir-path*, otherwise the returned path specification is the full file path of *file-path*.

Note that the current directory symbol, ‹`.`› (dot), is regarded as an absolute path. For example, ‹`.\MyFile.txt`› is regarded as an absolute path.

Examples

The following illustrations show how the **fCvtRelativePath** function can be used.

```
(1) fCvtRelativePath('C:\MyFolder\Other' 'Programs\TextFile.txt');
(2) fCvtRelativePath('C:\MyFolder\Other' 'C:\Files\TextFile.txt');
```

Example **(1)** returns the string ‹`C:\MyFolder\Other\Programs\TextFile.txt`› because the second argument is a relative path to the first argument.

Example **(2)** returns the string ‹`C:\Files\TextFile.txt`› because the second argument is an absolute path; the first argument is ignored. ♦

fDateTimeFormatted

fDateTimeFormatted *fmt-dt utc* → *dt-str*	@F_DATE_TIME_FORMATTED
fmt-dt	A string stack object.
utc	An integer stack object containing a logical boolean value.
dt-str	A string stack object.

Details

Returns a formatted date and time string (*dt-str*) of the current date and time based on a specified format (*fmt-dt*). If *utc* is `true`, the returned string represents the UTC ("Universal Time Coordinated") date and time (previously referred to as "Greenwich Mean Time" or GMT), otherwise it represents the local date and time.

The following table shows the date\time symbols and their meaning within the format string *fmt-dt*. Other symbols (eg: '`/`') are presented as given. Where a single digit is specified, only leading zero digits are suppressed; other non-zero digits are presented. For example, if the seconds is `20`, then ‹`[s]`› will display `20`; if the seconds is `3`, then ‹`[s]`› will display `3`, but ‹`[ss]`› will display `03`.

Desired Date and Time	Format Symbol
Year (four digits, last two digits)	`[yyyy]`, `[yy]`
Month (long name, short name, two digits, one digit)	`[MMMM]`, `[MMM]`, `[MM]`, `[M]`
Day (long name, short name, two digits, one digit)	`[dddd]`, `[ddd]`, `[dd]`, `[d]`
12 hour (two digits, one digit)	`[hh]`, `[h]`
24 hour (two digits, one digit)	`[HH]`, `[H]`
Minute (two digits, one digit)	`[mm]`, `[m]`
Second (two digits, one digit)	`[ss]`, `[s]`
Fraction of seconds (3 digits)	`[f]`
AM\PM (`A\P`, `AM\PM`, `a\p`, `am\pm`)	`[T]`, `[TT]`, `[t]`, `[tt]`

Examples

The following illustrations show how the **fDateTimeFormatted** function can be used.

```
(1) fDateTimeFormatted("Today is [dd]/[MM]/[yyyy] [HH]:[mm]:[ss]", false);
(2) fDateTimeFormatted("Today is [ddd] [dd]-[M]-[yy] [h]:[mm]:[s].[f] [tt]",
    true);
(2) fDateTimeFormatted("It is [dddd], day [d], in the month of [MMMM], in the
    year [yyyy] AD.", true);
```

Example **(1)** returns the current local date and time in a form such as ‹`Today is 20/05/2014 19:06:23`›.

Example **(2)** returns the current UTC date and time in a form such as ‹`Today is Tue 20-5-14 7:06:23.592 pm`›.

Example **(3)** returns the current UTC date and time in a form such as ‹`It is Tuesday, day 20, in the month of May, in the year 2014 AD.`›. ♦

fDelDuplStrs

fDelDuplStrs *in-seq* → *out-seq*	@F_DEL_DUPL_STRS
in-seq	A string sequence.
out-seq	A string sequence.

Details

Deletes duplicate elements of a string sequence (*in-seq*) returning a new string sequence (*out-seq*). Note that the original string sequence, *in-seq*, is not modified. All strings in the returned string sequence will be unique.

The function leaves the first one of the duplicate strings and removes the other duplicates.

Examples

The following illustrations show how the **fDelDuplStrs** function can be used.

```
(1) Seq := ["hello", "goodbye", "hello"]; RtnSeq := fDelDuplStrs(Seq);
(2) RtnSeq := fDelDuplStrs(fReadTextLines("MyTextFile.txt"));
```

Example **(1)** returns the new sequence ‹["hello", "goodbye"]› in `RtnSeq`, leaving the original sequence in `Seq` unmodified.

Example **(2)** assumes that the specified file exists, and removes duplicate text lines from an internal copy of the file data with no duplicate lines, returning the result in `RtnSeq`. Note that in this example, the **fReadTextLines** function returns a string sequence containing the data of the specified file. ♦

fExecETACStr

fExecETACStr *str*	@F_EXEC_ETAC_STR
str	A string stack object.

Details

Executes a string (*str*) containing ETAC text. The string is executed as *ETAC text script*, with the inclusion file TACGlobal.PTAC automatically included before the string.

fExecETACStr effectively executes the following in a new *ETAC session*.

```
[*::10ETAC-SOURCE-V1::*]
::include TACGlobal.PTAC
str;
```

Examples

The following illustrations show how the **fExecETACStr** function can be used.

```
(1) fExexETACStr("(3 + 5)");
(2) fExexETACStr("add2 3 5;");
(3) fExexETACStr("exec {add2 3 5;}");
```

Examples **(1)** to **(3)** return the value `8` on the object stack. ♦

fExtractInnerStr

fExtractInnerStr *in-str* → *out-str*	@F_EXTRACT_INNER_STR
in-str	A string stack object.
out-str	A string stack object.

Details

Extracts the substring (*out-str*) from a string (*in-str*) enclosed within brackets. *in-str* must begin with one of the bracket characters, ‹(›, ‹[›, ‹{›, and end with the corresponding bracket character, ‹)›, ‹]›, ‹}›, otherwise *out-str* will be the same as *in-str*. If *in-str* is delimited as said, *out-str* will contain the text in-between, but excluding, the bracket characters.

Examples

The following illustrations show how the **fExtractInnerStr** function can be used.

```
(1) fExtractInnerStr("[the string]");
(2) fExtractInnerStr("not delimited by brackets");
(3) fExtractInnerStr("[mismatched brackets)");
```

Example **(1)** returns ‹`the string`›.

Example **(2)** returns ‹`not delimited by brackets`› because *in-str* was not delimited by the appropriate brackets.

Example **(3)** returns ‹`[mismatched brackets)`› because the outer brackets to not correspond. ♦

fFloor

fFloor *num* → *floor*	@F_FLOOR
num	A decimal or integer stack object.
floor	A decimal or integer stack object.

Details

Returns (*floor*) the floor of a decimal or integral number (*num*). The floor of a number is the largest integer less than or equal to that number.

If *num* is an integer then *floor* will also be an integer equal to *num*. If *num* is a decimal then *floor* will be an integer if it is within the range of an integer (–2,147,483,648 to 2,147,483,647), otherwise it will be an integral decimal. ♦

fFormatStr

fFormatStr *fmt-str* *repl-seq* → *str*	@F_FORMAT_STR
fmt-str	A string stack object.
repl-seq	A numeric or string sequence, or a null stack object (`?`).
str	A string stack object.

Details

Replaces symbols within a format string (*fmt-str*), returning a formatted string (*str*). This function creates a new temporary local dictionary when processing *fmt-str*.

The symbols within *fmt-str* are of the form:

1. ‹%*n*%› where *n* is a positive integer. There can be more than one %*n*% for a particular *n*, but no *n* can exceed the number of elements in *repl-seq*. Each *n* is an index into *repl-seq*, which must be a numeric or string sequence. The string or number at that index (*n*) replaces all the %*n*% in *fmt-str*. Whitespaces must not exist between the percent characters and *n*.
2. ‹%*› is replaced with %. Note that % is special everywhere else, so an actual literal % must be represented as ‹%*›.
3. ‹%(*expression*)%› where (*expression*) is an ETAC expression, which must return a number or string when activated. The symbol is replaced by the string form of the returned value.
4. ‹%{*procedure*}%› where {*procedure*} is an ETAC procedure, which must return a number or string when activated. The symbol is replaced by the string form of the returned value.
5. ‹%*variable*%› where *variable* is an ETAC variable, which must return a number or string when activated. The symbol is replaced by the string form of the returned value. Whitespaces must not exist between the percent characters and *variable*. Note that the value of the dictionary item represented by *variable* can be a procedure, which is executed, returning a number or string.

Any number of the above symbols can be used in the same *fmt-str*. The command cpy can be used within *expression* (3) and *procedure* (4) above. cpy is identical to the ETAC command **copy_top**. If symbol (1) is not used, then the second argument (*fmt-str*) to the function is ignored (it must be the null stack object).

If a symbol is correct but the resulting value could not be converted to a string, then the symbol is replaced by ‹?*text*?›, where *text* is the text between the percent characters of the symbol. If a symbol could not be processed then it remains as is.

The function returns the modified string *str*.

Examples

The following illustrations show how the **fFormatStr** function can be used. The symbols mentioned above are highlighted (pink) in the examples.

```
(1) fFormatStr("The %1% chased his %2%." ["dog", "tail"]);
(2) fFormatStr("Some people %1% %1% %1% themselves 10%* of the time %1%edly"
    ["repeat"]);
(3) Var := "programmers"; fFormatStr("Hello %Var%" ?);
(4) Var := {("pro" + "grammers");}; fFormatStr("Hello %Var%" ?);
(5) fFormatStr('Hello %("pro" + "grammers")%' ?);
(6) fFormatStr('Hello %{add2 "pro" "grammers";}%' ?);
(7) fFormatStr("The value of %1% times %2% is %(&@ V :- cpy (2 * 3))%. That
    value %V% is %4%." ["two", 3, "incorrect", "correct"]);
```

Example **(1)** returns ‹The dog chased his tail.›.

Example **(2)** returns ‹Some people repeat repeat repeat themselves 10% of the time repeatedly›.

Examples **(3)** to **(6)** return ‹Hello programmers›.

Example **(7)** returns ‹The value of two times 3 is 6. That value 6 is correct.›. The variable V is allocated temporarily within **fFormatStr**. Note the use of the special cpy command, which is the same as the **copy_top** ETAC command. ♦

fFromWinCC

fFromWinCC *wcc* → *char*	@F_FROM_WIN_CC
wcc	A non-negative integer stack object.
char	A string stack object.

Details

Converts a Windows-1252 character code (*wcc*) to a string (*char*) containing the corresponding character. *wcc* must be a non-negative integer less than 256, otherwise the consequence is undefined. If *wcc* is zero, then *char* will be empty. ♦

fGetKWArgs

fGetKWArgs *tmpl* *arg-str* *sep* → *bool* *str-seq*	@F_GET_KW_ARGS
tmpl	A string stack object, or a string sequence.
arg-str	A string stack object.
sep	A string stack object, or a null stack object (?).
bool	An integer stack object containing a logical boolean value.
str-seq	A string sequence.

Details

Processes keywords and their arguments (*arg-str*), based on a keyword template (*tmpl*), into a string sequence tree (*str-seq*). Note that this function operates in the same way as the ETAC command **kw_args**.

sep is a string containing three UCS-2 (BMP Unicode scalar value) characters. The left-most character determines the character for separating the parameters in the keyword template specified in *tmpl*. The middle character determines the source argument separators in the argument string (*arg-str*). That character cannot be a whitespace. The right-most character must be a zero character (0). For example, the string ‹,#0› indicates that the parameters specified in *tmpl* are separated by commas (the default), and the arguments in *arg-str* are separated by a hash character. The default is effectively ‹,,0›, and is indicated by a null stack object (?) for *sep*.

tmpl is either a string indicating a single keyword template, or a sequence of strings each of which is part of a nested keyword template. The first sequence element specifies the main group of template blocks; each other element specifies a keyword block. A string value for *tmpl* is effectively a sequence containing that string value.

Important Note

If any element of *tmpl* does not have a valid syntax, the consequence is unpredictable. **fGetKWArgs** does not cater for elements with an invalid syntax.

arg-str is the source string to be parsed, consisting of keywords and their arguments. ETAC comments within *arg-str* are logically replaced with one space, unless the comments are within a pair of double quotes. Backslashes (‹\›) in *arg-str* that are outside of string blocks are ignored and the character following a backslash is accepted literally. Escaped ETAC comments outside of string blocks are retained, as in this example, ‹KW=argument \[*comment retained*\]›. The backslashes are automatically removed, leaving ‹KW=argument [*comment retained*]› as the effective *arg-str*. Without the backslashes in the example, the comment is replaced with a single space. *arg-str* can include double-angle quoted substrings (the :!KA_ANGLE_QUOTES: flag is automatically applied), so the example above can be presented as ‹KW=argument «[*comment retained*]»› to retain the comment.

bool indicates whether *arg-str* has matched the keyword template in *tmpl*. If *bool* is `true`, the match was successful, and the parsed *arg-str* will be contained in the returned output tree (*str-seq*). If *bool* is `false`, the match failed, and *str-seq* will contain a sequence of strings describing the reasons for the failure.

str-seq is the output tree containing nested sequences for each matched block corresponding to the keyword template in *tmpl*. Each matched and parsed block consists of a sequence containing one or more subsequences. Each subsequence contains string elements. The first element in the subsequence is the matched keyword, and the subsequent elements are the matched arguments or a matched and parsed block.

For full information on the keyword-argument system see **Appendix A: Keyword-arguments Specification** in the document "*The Official ETAC Programming Language*" (ETACProgLang(Official).pdf).

Example

The following illustration shows how the **fGetKWArgs** function can be used.

(1)	`fGetKWArgs("{//A(#a1)/k/C(x)}{/D($a2)/-e(#a3,?)}" "-ea,b C -e c" ?);`

Example **(1)** returns `true` followed by the sequence
‹`[["C", "x"], ["-e", "a", "b", "c"]]`› as the second top stack object.

Additional Information

See **Appendix A: Keyword-arguments Specification** in the document "*The Official ETAC Programming Language*" (ETACProgLang(Official).pdf). ♦

fGetKWSyntax

fGetKWSyntax *tmpl* *sep* → *sntx-str*	@F_GET_KW_SYNTAX
tmpl	A string stack object, or a string sequence.
sep	A string stack object, or a null stack object (`?`).
sntx-str	A string stack object.

Details

Gets the keyword-arguments syntax (*sntx-str*) of a keyword template (*tmpl*). *sntx-str* will contain a string showing the user-friendly syntax based on *tmpl*.

tmpl and *sep* are as defined for the function **fGetKWArgs**, except that for *sep* (if it is a string) the middle character must be a zero character (`0`), and the third character determines the character for separating the options in the returned syntax (*sntx-str*). The default is effectively ‹`,0\`›, and indicated by a null stack object (`?`) for *sep*. (Note that in ETAC, a backslash character in a double-quoted string is represented as two backslashes or a backslash followed by a space.)

Example

The following illustration shows how the **fGetKWSyntax** function can be used.

(1)	`fGetKWSyntax("{//A(#a1)/k/C(x)}{/D($a2)/-e(#a3,?)}" "00|");`

Example **(1)** returns the syntax string ‹`<A a1|k|C> [D {a2}|-e a3, ...]`›.

Additional Information

fGetKWArgs ♦

fGetMemSize

fGetMemSize *mem* → *size*	@F_GET_MEM_SIZE
mem	A memory stack object.
size	An integer stack object.

Details

Returns (*size*) the byte size of the usable data in a memory stack object (*mem*). *size* could be a negative integer representing the equivalent positive integer in two's complement format. ♦

fGetStrU

fGetStrU *str* *offset* *len* → *out-str*	@F_GET_STR_U
str	A string stack object.
offset	A non-negative integer stack object.
len	A non-negative integer stack object.
out-str	A string stack object.

Details

Returns (*out-str*) the middle substring (*offset*, *len*) of a string (*str*). *offset* and *len* are in *u-char* character units.

offset is a zero-based *u-char* character offset into *str*. If *offset* indicates a character beyond the last *u-char* character of *str*, then an empty string will be returned by the function.

len is the maximum number of *u-char* characters to be obtained from *str* beginning at *offset*. If *len* exceeds the remaining characters of *str*, then only the remaining characters are obtained. If *len* is zero, then an empty string will be returned by the function.

out-str is a substring of *str* beginning at *u-char* character *offset* with *u-char* character length up to *len*.

Examples

The following illustrations show how the **fGetStrU** function can be used.

```
(1) fGetStrU("hello-ha" 5 1);
(2) fGetStrU("hello-ha" 6 4);
(3) fGetStrU("hello-ha" 8 4);
(4) fGetStrU("thumbs\#1F44D#up" 6 2);
(5) fGetStrU("thumbs\#1F44D#up" 7 2);
```

Example **(1)** returns the string ‹-›.

Example **(2)** returns the string ‹ha›.

Example **(3)** returns an empty string because *offset* is beyond the last character of *str*.

Example **(4)** returns the string equivalent of ‹\#1F44D#u›. Note that the Unicode supplementary plane code point U+1F44D (Thumbs Up Sign) is internally represented as a surrogate pair, but is only one *u-char* character wide. Because *len* (2) is the *u-char* character length, both *w-chars* (surrogate pairs) of the character at *offset* 6 and the following character (u), are obtained.

Example **(5)** returns the string ‹up› because it begins at *u-char* character offset 7 of *str*. ♦

fGetWindowsDir

fGetWindowsDir → *dir-str* | ?	@F_GET_WINDOWS_DIR
dir-str	A string stack object.

Details

Get the full path of the system `Windows` directory (*dir-str*) or a null stack object (`?`) if that directory could not be obtained. *dir-str* will typically contain ‹`C:\Windows`›.

Illegal UTF-16 characters (ie: unpaired Unicode surrogate code points) in *dir-str* will be replaced with '`?`' (question mark). ♦

fHexToInt

fHexToInt *str* → *int* | ?	@F_HEX_TO_INT
str	A string stack object.
int	An integer stack object.

Details

Converts the first eight characters or less of a hexadecimal string (*str*) to an equivalent integer (*int*).

If *str* is an empty string then *int* will be zero. If *str* contains eight or less characters and not all of those characters are hexadecimal text, then a null stack object will be returned. The hexadecimal text characters are '`0`' to '`9`', '`A`' to '`F`', and '`a`' to '`f`'.

Examples

The following illustrations show how the **fHexToInt** function can be used.

```
(1) fHexToInt("3A5f");
(2) fHexToInt("45fa8Bd2iy");
```

Example **(1)** returns `14943`, which is equal to $3A5F_{16}$.

Example **(2)** returns `1174047698`, which is equal to $45FA8BD2_{16}$. ♦

fIndentLines

fIndentLines *str-seq num-pos pad eolchrs*	@F_INDENT_LINES
str-seq	A string sequence.
num-pos	A non-negative integer stack object.
pad	A string stack object.
eolchrs	A string stack object.

Details

Indents all text lines in a string sequence (*str-seq*) the specified number of positions (*num-pos*) filled with the specified *w-char* character (*pad*). Indentation also applies to the sequence elements containing specified EOL (end-of-line) characters (*eolchrs*).

An element of *str-seq* could contain more than one text line, each separated by the string in *eolchrs*. For example, the string element `"line 1\r\nline 2\r\nline 3"` (ie: ‹`line 1`$^{C}{}_{R}{}^{L}{}_{F}$`line 2`$^{C}{}_{R}{}^{L}{}_{F}$`line 3`›) contains three text lines if *eolchrs* is `"\r\n"` (ie: $^{C}{}_{R}{}^{L}{}_{F}$).

str-seq will have each text line within each string element modified (indented) by this function.

num-pos is a non-negative integer indicating the number of positions to indent the text lines in *str-seq*.

pad is a string, but only the first character, which must be a UCS-2 (BMP Unicode scalar value) character, is used to pad the indentation. *pad* will typically be a space character. If *pad* is an empty string then no indentation will occur.

eolchrs is a string that separates text lines within *str-seq*. If each element of *str-seq* is a single text line, then *eolchrs* should be an empty string.

Examples

The following illustrations show how the **fIndentLines** function can be used.

```
(1) Seq := ["First line", "line 1\nline 2\nline 3"];
    fIndentLines(Seq 3 "*" "\n");
(2) Seq := ["First line", "line 1::line 2::line 3::"];
    fIndentLines(Seq 3 "!" "::");
(3) Seq := ["\r\n\r\n"];
    fIndentLines(Seq 3 " " "\r\n");
```

In example **(1)**, `Seq` will end up containing the two elements ‹`***First line`› and ‹`***line 1␊***line 2␊***line3`›.

In example **(2)**, `Seq` will end up containing the two elements ‹`!!!First line`› and ‹`!!! line 1::!!!line 2::!!!line3::!!!`›.

In example **(3)**, `Seq` will end up containing the single element ‹␠␠␠␍␊␠␠␠␍␊␠␠␠›, that is, three text lines each containing three spaces. ♦

fIsFileWritable

fIsFileWritable *file-path* → *str*	@F_IS_FILE_WRITABLE
file-path	A string stack object.
str	A string stack object.

Details

Checks that a file at a specified file path (*file-path*) is writeable if it exists. Returns an error message (*str*) if it is not writeable, otherwise returns an empty string.

If the file does not exist at the specified path, *str* will be an empty string. ♦

fIsOnlyDirPath

fIsOnlyDirPath *path* → *bool*	@F_IS_ONLY_DIR_PATH
path	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a path specification (*path*) is a directory path only. If the last character of *path* is a forward slash or backslash, the function returns `true`, otherwise it returns `false`. ♦

fIsOnlyFileName

fIsOnlyFileName *path* → *bool*	@F_IS_ONLY_FILE_NAME
path	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a file path specification (*path*) contains only a file name (and extension). If the file name and extension part of *path* is equal to *path*, the function returns `true`, otherwise it returns `false`. ♦

fIsRelativePath

fIsRelativePath *path* → *bool*	@F_IS_RELATIVE_PATH
path	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a file path specification (*path*) is a relative path. If *path* does not contain a drive part and does not begin with a current directory symbol, ‹.› (dot), the function returns `true`, otherwise it returns `false`.

Note that the current directory symbol, ‹.› (dot), is regarded as an absolute path. For example, ‹`.\MyFile.txt`› is regarded as an absolute path.

Examples

The following illustrations show how the **fIsRelativePath** function can be used.

```
(1) fIsRelativePath("MyPath\\MyFile.txt");
(2) fIsRelativePath('C:\MyPath\MyFile.txt');
```

Example **(1)** returns `true`, while example **(2)** returns `false`. ♦

fIsStrDblQuoted

fIsStrDblQuoted *str* → *bool*	@F_IS_STR_DBL_QUOTED
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is delimited by double-quote characters (`"`). If the first and last characters of *str* are double-quote characters (U+0022), then the function returns `true`, otherwise it returns `false`. ♦

fIsStrDec

fIsStrDec *str* → *bool*	@F_IS_STR_DEC
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of a decimal number (including exponential notation). If *str* is of the form

[[{+}|-]*digits*].*digits*[(e|E)[{+}|-]*digits*] or

[{+}|-]*digits*(e|E)[{+}|-]*digits*,

where *digits* is one or more decimal digits, the function returns `true`, otherwise it returns `false`. ♦

fIsStrInt

fIsStrInt *str* → *bool*	@F_IS_STR_INT
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of an integer. If *str* is of the form

[{+}|-]*digits*,

where *digits* is one or more decimal digits, the function returns `true`, otherwise it returns `false`. ♦

fIsStrNegInt

fIsStrNegInt *str* → *bool*	@F_IS_STR_NEG_INT
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of a negative integer. If *str* is of the form

-*digits*,

where *digits* is one or more decimal digits, the function returns `true`, otherwise it returns `false`. Note the minus sign before *digits*. ♦

fIsStrPosInt

fIsStrPosInt *str* → *bool*	@F_IS_STR_POS_INT
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of a positive integer. If *str* is of the form

[{+}]*digits*,

where *digits* is one or more decimal digits, the function returns `true`, otherwise it returns `false`. Note the optional plus sign before *digits*. ♦

fIsStrZeroInt

fIsStrZeroInt *str* → *bool*	@F_IS_STR_ZERO_INT
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) is in the form of a zero integer. If *str* is of the form

[+|-]*digits*,

where *digits* is one or more zero digits, the function returns `true`, otherwise it returns `false`. For example, ‹`-000`› returns `true`. ♦

fLambda

fLambda *b-pars* *vars* *proc* → *fnt*	@F_LAMBDA
b-pars	A string sequence.
vars	A string sequence.
proc	A procedure.
fnt	An ETAC function.

Details

Creates a general lambda abstraction function for any predefined function or procedure. Lambda functions in programming are (supposed to be) based on the mathematical lambda calculus, which essentially involves mappings from mathematical expressions to other mathematical expressions.

The terms ‘bound parameter’ and ‘free parameter’ correspond to the terms ‘bound variable’ and ‘free variable’, respectively, in lambda calculus. The terms ‘abstraction’ and ‘application’ also correspond to similar terms in lambda calculus. To understand lambda functions in programming it is useful to understand the essence of mathematical lambda calculus.

b-pars is a sequence of *bound parameters* of the returned *abstraction* function (*fnt*). The sequence consists of quoted parameter names, for example `["pPar3", "pPar1"]`.

vars is a sequence of the remaining *free parameters* of the target function or procedure. The sequence consists of quoted parameter names, for example `["pPar2", "pPar4"]`. *vars* can be an empty sequence.

proc is a procedure containing the call to the target function or procedure involving the parameters specified in the first two sequences, for example `{MyFnt(pPar1 pPar2 pPar3 pPar4);}` or `{(+ pPar1 pPar3 pPar2 pPar4);}`.

This function returns a lambda *abstraction* function (*fnt*) which is defined with the parameters specified in *b-pars* in the given order. That returned function is later called with values that bind those parameters, itself returning a function (a lambda *application* function) which is defined with the parameters specified in *vars* (the *free parameters*) and with the parameters in *b-pars* already bound.

The lambda *abstraction* function returned by the call to ‹`fLambda([`*b*`, ⋯] [`*f*`, ⋯] {...})`› is defined as being equivalent to ‹`fnt:(`*b* `⋯){@Localise(fnt:(`*f* `⋯){...});}`›. When that returned function itself is called with the arguments that correspond to the *bound parameters* (*b* ⋯), it returns a function defined as being equivalent to the localised function ‹`fnt:(`*f* `⋯){...}`›. That last returned function (the *application* function) is then called with arguments that correspond to the *free parameters* (*f* ⋯). Note that **fLambda** is only a convenience for producing an *abstraction* function; an *abstraction* function can be defined directly without creating it via **fLambda**.

NOTE that the *bound parameters* in the lambda *abstraction* function can be bound to any TAC object, including functions, procedures, and operators, for an appropriately constructed *proc* value.

Examples

The following illustrations show how the **fLambda** function can be used.

(1)	`fLambda(["pPar3", "pPar1"] ["pPar2", "pPar4"] {MyFnt(pPar1 pPar3 pPar2 pPar4);});`
(2)	`fLambda(["pConst"] ["pVar"] {(pConst + pVar);});`
(3)	`fLambda(["pConst"] [] {("Statement: " + pConst);});`
(4)	`Abstraction := fLambda(["pConst"] ["pVar"] {(pConst + pVar);});` `Application := Abstraction(5);`

Example **(1)** returns a function internally defined as ‹`fnt:(pPar3 pPar1){@Localise(fnt:(pPar2 pPar4){MyFnt(pPar1 pPar3 pPar2 pPar4);});}`›. The *bound parameters* are `pPar3` and `pPar1`. The *free parameters* are `pPar2` and `pPar4`.

Example **(2)** returns a function internally defined as ‹`fnt:(pConst){@Localise(fnt:(pVar) {(pConst + pVar);});}`›. The *bound parameter* is `pConst`. The *free parameter* is `pVar`.

Example **(3)** returns a function internally defined as ‹`fnt:(pConst){@Localise(fnt:() {("Statement: " + pConst);});}`›. The *bound parameter* is `pConst`. There are no *free parameters*.

In each of the three examples above, the returned function needs to be called with values for its *bound parameters*; the call then returns another function with those parameters already bound. That other function then needs to be called with values for the *free parameters* of the original lambda function.

In example **(4)**, the function `Abstraction` contains the returned function mentioned in example (2), where the *bound parameter*, `pConst`, is bound to `5` by the call `Abstraction(5)`. `Abstraction(5)` returns an *application* function assigned to `Application`. The function `Application` is equivalently defined as ‹`fnt:(pVar){(5 + pVar);}`›, so `Application(10)` returns `15`, and `Application(3)` returns `8` (the arguments `10` and `3` are the values for the *free parameter* `pVar`).

In practice, **fLambda** is rarely used except in cases where many different instances of a particular function need to be made with different values of the same subset of the parameters of that function.

Other Information

fLambdaApp ♦

fLambdaApp

fLambdaApp *b-pars vars call* → *fnt*	@F_LAMBDA_APP
b-pars	A string sequence.
vars	A string sequence.
call	A procedure.
fnt	An ETAC function.

Details

Creates a general lambda application function for any predefined function or procedure. See the entry fLambda before reading this entry.

b-pars is a sequence of *bound parameters* and values of the target function or procedure. The sequence consists of pairs of quoted parameter names with their values, for example ‹`["pPar3", 10, "pPar1", "string"]`› (`pPar3` is to be bound to `10`, and `pPar1` is to be bound to `"string"`).

vars is a sequence of the remaining *free parameters* of the target function or procedure. The sequence consists of quoted parameter names, for example ‹`["pPar2", "pPar4"]`›. *vars* can be an empty sequence.

call is a procedure containing the call to the target function or procedure involving the parameters specified in the first two sequences, for example ‹`{MyFnt(pPar1 pPar2 pPar3 pPar4);}`› or ‹`{(+ pPar1 pPar3 pPar2 pPar4);}`›.

This function returns a lambda *application* function (*fnt*) which is defined with the parameters specified in *vars* (the *free parameters*) and with the parameters in *b-pars* already bound.

The lambda *application* function returned by the call to ‹fLambdaApp([*b*, *v*, ⋯] [*f*, ⋯] {**...**})› is defined as being equivalent to the localised function ‹fnt:(*f* ⋯){**...**}› with each parameter *b* already bound to its corresponding argument *v*. That returned function (the *application* function) is then called with arguments that correspond to the *free parameters* (*f* ⋯). Note that calling ‹fLambdaApp([*b*, *v*, ⋯] [*f*, ⋯] {**...**})› is equivalent to executing ‹@Call fLambda([*b*, ⋯] [*f*, ⋯] {**...**}) *v* ⋯›. So, **fLambdaApp** is only a convenience for producing an *application* function directly.

NOTE that the values of the *bound parameters* in *b-pars* can be any TAC object, including functions, procedures, and operators, for an appropriately constructed *call* value.

Examples

The following illustrations show how the **fLambdaApp** function can be used.

(1)	**fLambdaApp**`(["pPar3", 10, "pPar1", "string"] ["pPar2", "pPar4"] {MyFnt(pPar1 pPar2 pPar3 pPar4);});`
(2)	**fLambdaApp**`(["pConst", 5] ["pVar"] {(pConst + pVar);});`
(3)	**fLambdaApp**`(["pConst", "Statement: "] ["pVar"] {(pConst + pVar);});`
(4)	`Application :=` **fLambdaApp**`(["pConst", 5] ["pVar"] {(pConst + pVar);});`
(5)	`Application := @Call fLambda(["pConst"] ["pVar"] {(pConst + pVar);}) 5;`

Example **(1)** returns a function equivalently defined as ‹`fnt:(pPar2 pPar4){MyFnt("string" pPar2 10 pPar4);}`›. The *free parameters* are `pPar2` and `pPar4`.

Example **(2)** returns a function equivalently defined as ‹`fnt:(pVar) {(5 + pVar);}`›. The *free parameter* is `pVar`.

Example **(3)** returns a function equivalently defined as ‹`fnt:(pVar) {("Statement: " + pVar);}`›. The *free parameter* is `pVar`.

In each of the three examples above, the *bound parameters* of the returned function are already bound to their corresponding values. That returned function then needs to be called with values for the *free parameters* (*vars*) specified in the original **fLambdaApp** function call.

In examples **(4)** and **(5)**, the function `Application` contains the same function as the returned function defined at example (2), so `Application(10)` returns `15`, and `Application(3)` returns `8` (the arguments `10` and `3` are the values for the *free parameter* `pVar`).

fLambdaApp is typically used instead of **fLambda** when there needs to be only one instance of binding the *bound variables* for a given target function or procedure (*call*).

Additional Information

fLambda ♦

fLinesToMem

fLinesToMem *str-seq* → *mem*	@F_LINES_TO_MEM
str-seq	A string sequence.
mem	A memory stack object.

Details

Converts from a string sequence (*str-seq*) to a memory object (*mem*) with EOL characters.

str-seq is a string sequence, and each of its elements is appended to a new memory object (*mem*) with the EOL (end-of-line) characters ‹CRLF›.

Example

The following illustration shows how the **fLinesToMem** function can be used.

```
(1) fLinesToMem(["Line 1", "Line 2", "Line 3"]);
```

Example **(1)** returns a memory object containing ‹Line 1CRLFLine 2CRLFLine3CRLF›. ♦

fLinesToStr

fLinesToStr *str-seq* → *str*	@F_LINES_TO_STR
str-seq	A string sequence.
str	A string stack object.

Details

Converts from a string sequence (*str-seq*) to a string (*str*) with EOL characters.

str-seq is a string sequence, and each of its elements is appended to a string (*str*) with the EOL (end-of-line) characters ‹CRLF›.

Example

The following illustration shows how the **fLinesToStr** function can be used.

```
(1) fLinesToStr(["Line 1", "Line 2", "Line 3"]);
```

Example **(1)** returns the string ‹Line 1CRLFLine 2CRLFLine3CRLF›. ♦

fMatchString

fMatchString *pat* *str* → *bool*	@F_MATCH_STRING
pat	A string stack object, or a string sequence.
str	A string stack object.
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a string (*str*) matches a *pattern string* and possibly sub-patterns (*pat*).

pat is a *pattern string* or a sequence containing *pattern strings* which is\are matched by the whole of *str*. The syntax for the strings in *pat* is as indicated under the heading **Additional Information**, except that *pat* cannot contain blocks of the form ‹<*n*...>›, where *n* is an integer from 0 to 9.

If *pat* is a sequence, the second and subsequent elements of that sequence contain custom *pattern strings* identified by the ‹<pr>› special characters in the elements of *pat*. The first custom pattern in *pat* (the second element of *pat*) is custom pattern number 0 (ie: pattern represented by <p**0**>); the next custom pattern in *pat* (the third element of *pat*) is custom pattern number 1 (ie: pattern represented by <p**1**>), and so on.

Important Note

If *pat* or any element of it (if *pat* is a sequence) does not have a valid syntax, the consequence is unpredictable. **fMatchString** does not cater for patterns with an invalid syntax.

str is the string to be matched against *pat*.

bool will be `true` if all of *str* matches *pat*, otherwise it will be `false`.

Examples

The following illustrations show how the **fMatchString** function can be used.

```
(1) fMatchString("%%{$?%%`d}%?" "there are 120 MINUTES in 2 HOURS");
(2) fMatchString(["%%{$?[{<p0>}{<p1>}]}%?", "%%`u", "%%`d"] "there are 120
    MINUTES in 2 HOURS");
```

Example **(1)** returns `true` because the whole string (second argument) matches the *pattern string* (first argument).

In example **(2)**, <p0> represents the pattern ‹`%%`u`›, and <p1> represents the pattern ‹`%%`d`›. The function returns `true` because the whole string (second argument) matches the sequence of *pattern strings* (first argument).

Additional Information

See **Pattern String Matching** under chapter 3 of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf.

Other Information

fParseString ♦

fMemToHexChars

fMemToHexChars *offset len mem* → *str*	@F_MEM_TO_HEX_CHARS
offset	An integer stack object.
len	An integer stack object.
mem	A memory stack object.
str	A string stack object.

Details

Converts a specified portion (*offset*, *len*) of a memory object (*mem*) into a hexadecimal string (*str*).

offset is interpreted as a positive integer or zero indicating a zero-based byte offset into *mem*.

len is interpreted as a positive integer or zero indicating the amount of bytes relative to *offset* to convert.

mem is the memory stack object. The content of *mem* remains unchanged.

str is a hexadecimal string representing the specified portion of memory, or an empty string if an error occurs. The hexadecimal text characters are '`0`' to '`9`' and '`A`' to '`F`'. Note that the hexadecimal characters 'a' to 'f' will be in uppercase.

Note that *offset* plus *len* need not be within the bounds of the usable memory within *mem*.

Examples

The following illustrations show how the **fMemToHexChars** function can be used.

```
(1) fMemToHexChars(2 3 &0h0122033d4F09);
(2) fMemToHexChars(0 4 &0h012203);
(3) fMemToHexChars(20 4 &0h010203);
```

Example **(1)** returns the string ‹033D4F›.

Example **(2)** returns the string ‹012203›.

Example **(3)** returns an empty string because *offset* is outside the bounds of the usable memory within *mem*. ♦

fMid

fMid *str offset len* → *out-str*	@F_MID
str	A string stack object.
offset	A positive integer stack object.
len	A positive integer stack object.
out-str	A string stack object.

Details

Returns (*out-str*) the middle substring (*offset*, *len*) of a string (*str*). *offset* and *len* are in *w-char* character units.

offset is a zero-based *w-char* character offset into *str*. If *offset* indicates a character beyond the last character of *str*, then *out-str* will be an empty string.

len is the maximum number of *w-char* characters to be obtained from *str* beginning at *offset*. If *len* exceeds the remaining characters of *str*, then only the remaining characters are obtained.

out-str is a substring of *str* beginning at *w-char* character *offset* with *w-char* character length up to *len*, or is an empty string. Note that substring indicated by *offset* and *len* must be a well-formed Unicode substring, otherwise an *error event* will occur.

Examples

The following illustrations show how the **fMid** function can be used.

```
(1) fMid("hello-ha" 5 1);
(2) fMid("hello-ha" 6 4);
(3) fMid("hello-ha" 8 4);
(4) fMid("thumbs\#1F44D#up" 6 2);
(5) fMid("thumbs\#1F44D#up" 7 2); [* An error event will occur. *]
```

Example **(1)** returns the string ‹-›.

Example **(2)** returns the string ‹ha›.

Example **(3)** returns a null stack object because *offset* is beyond the last character of *str*.

Example **(4)** returns the string equivalent of ‹\#1F44D#›. Note that the Unicode supplementary plane code point U+1F44D (Thumbs Up Sign) is internally represented as a surrogate pair, and is two *w-char* characters wide. Because *len* (2) is the *w-char* character length, both *w-chars* (surrogate pairs) of the character at *offset* 6 are obtained, forming a well-formed Unicode substring.

In example **(5)**, an *error event* occurs because the specified substring is not a well-formed Unicode substring (the *offset* 7 indicates the second *w-char* of the surrogate pair of ‹\#1F44D#›). ♦

fParseString

fParseString *pat str* → *str-seq* | ?	@F_PARSE_STRING
pat	A string stack object, or a string sequence.
str	A string stack object.
str-seq	A string sequence.

Details

Parses a string (*str*) based on a *pattern string* and possibly sub-patterns (*pat*), returning a string sequence (*str-seq*) corresponding to the parsed substrings of the string (*str*).

pat is a *pattern string* or a sequence containing *pattern strings* which is\are matched by the whole of *str*. Substrings within *str* matching 'blocks' within *pat* are captured into *str-seq* as strings. A *block* is of the form ‹<*n*...>›, where *n* is either 0 (zero) or an integer from 1 to 9. *Blocks* can be nested. *pat* can be an empty string, resulting in *str-seq* being an empty sequence. The syntax for the strings in *pat* is as indicated under the heading **Additional Information**.

If *pat* is a sequence, the second and subsequent elements of that sequence contain custom *pattern strings* identified by the ‹<pr>› special characters in the elements of *pat*. The first custom pattern in *pat* (the second element of *pat*) is custom pattern number 0 (ie: pattern represented by <p**0**>); the next custom pattern in *pat* (the third element of *pat*) is custom pattern number 1 (ie: pattern represented by <p**1**>), and so on.

> **Important Note**
>
> If *pat* or any element of it (if *pat* is a sequence) does not have a valid syntax, the consequence is unpredictable. **fParseString** does not cater for patterns with an invalid syntax.

str is the string to be parsed.

str-seq can be in either one of two formats. ‹<*n*...>› *blocks* are used in *pat* to produce the contents of *str-seq* matching those *blocks*. **Format 1**: *pat* contains only ‹<0...>› *blocks*. In that case, *str-seq* will be a flat string sequence. **Format 2**: *pat* contains only ‹<*m*...>› *blocks*, where *m* is an integer from 1 to 9, inclusive. In that case, *str-seq* will contain one level of string subsequences. *Block m* corresponds to element *m* of *str-seq* (note that *m* cannot be 0 in this case). *str-seq* will contain as many elements as the maximum *block* number in *pat*; omitted *block* numbers in *pat* will correspond to empty subsequences in *str-seq*. If a ‹<*m*...>› *block* exists in *pat* but there are no matches for that *block* then the corresponding subsequence in *str-seq* will be empty.

If the match fails completely, or no *blocks* exists in *pat*, then a null stack object (?) will be returned.

Examples

The following illustrations show how the **fParseString** function can be used.

```
(1) fParseString("%%{$?<0%%`d>}%?" "there are 120 MINUTES in 2 HOURS");
(2) fParseString(["%%{$?[{<p0>}{<p1>}]}%?", "<1%%`u>", "<3%%`d>"] "there are
    120 MINUTES in 2 HOURS");
```

Example **(1)** obtains the strings of all runs of one or more digits. The function returns the sequence ‹["120", "2"]›.

Block **1** in the example **(2)** obtains the strings of all runs of one or more uppercase characters; *block* 2 does not exist so it corresponds to the empty sequence in *str-seq*; and *block* **3** obtains the strings of all runs of one or more digits. <p0> represents the pattern <1%%`u>, and <p1> represents the pattern <3%%`d>. The function returns the sequence ‹[["MINUTES", "HOURS"], [], ["120", "2"]]› corresponding to the three *blocks* in *pat*.

Additional Information

See **Pattern String Matching** under chapter 3 of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf.

Other Information

fMatchString ♦

fPathExists

fPathExists *path type* → *bool*	@F_PATH_EXISTS
path	A string stack object.
type	An integer stack object, or a null stack object (?).
bool	An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether a specified type of disk entity (*type*) exists for a path specification (*path*).

type can be any one of the following: :#FP_PATH_FILE: (the entity is a file), :#FP_PATH_DIR: (the entity is a directory), :#FP_PATH_VOL: (the entity is a volume), or ? (the entity is a file or directory).

bool is true if *path* represents the entity specified by *type*, otherwise it is false. ♦

fPutStrU

fPutStrU *str offset len rep-str* → *out-str*	@F_PUT_STR_U
str	A string stack object.
offset	A non-negative integer stack object.
len	A non-negative integer stack object.
rep-str	A string stack object.
out-str	A string stack object.

Details

Replaces a substring at a *u-char* character offset and length (*offset*, *len*) in a given string (*str*) with a string (*rep-str*), returning the modified string (*out-str*).

offset is a zero-based *u-char* character offset into *str*. If *offset* indicates a character beyond the last character of *str*, then *out-str* will be the same as *str*.

len is the maximum number of *u-char* characters to be replaced in *str* beginning at *offset*. If *len* exceeds the remaining characters of *str*, then only the remaining characters are replaced.

rep-str is the string that replaces the substring indicated by *offset* and *length*.

out-str is the modified string after replacement.

Examples

The following illustrations show how the **fPutStrU** function can be used.

```
(1) fPutStrU("hello-ha" 5 1 " ");
(2) fPutStrU("hello-ha" 6 4 "*%");
(3) fPutStrU("hello-ha" 8 4 "*%");
(4) fPutStrU("hello-ha" 1 3 "");
(5) fPutStrU("thumbs\#1F44D#up" 6 2 "*");
(6) fPutStrU("thumbs\#1F44D#up" 7 2 "\#1F34F~green apple#**");
```

Example **(1)** returns the string ‹`hello ha`›.

Example **(2)** returns the string ‹`hello-*%`›.

Example **(3)** returns the string ‹`hello-ha`› because *offset* is beyond the last character of *str*.

Example **(4)** returns the string ‹`ho-ha`›.

Example **(5)** returns the string ‹`thumbs*p`›. Note that the Unicode supplementary plane code point U+1F44D (Thumbs Up Sign) is internally represented as a surrogate pair, but is only one *u-char* character wide. Because *len* (`2`) is the *u-char* character length, both *w-chars* (surrogate pairs) of the character at *offset* `6`, and the following character (`u`), are replaced.

Example **(6)** returns the equivalent of the string ‹`thumbs\#1F44D#\#1F34F#**`› because the substring being replaced begins at *u-char* character *offset* `7`. Note that the text ‹`~green apple`› is ignored by ETAC (such text is an example of an "in-string comment"). ♦

fQuickSort

fQuickSort *seq lo-idx hi-idx*	@F_QUICK_SORT
seq	A sequence.
lo-idx	A integer stack object.
hi-idx	A integer stack object.

Details

Sorts a sequence (*seq*) in place between elements at indices *lo-idx* and *hi-idx* using the "quick sort" algorithm. The elements of *seq* must be comparable. The order of the elements of *seq* may be modified by this function.

If *lo-idx* or *hi-idx* is out of range, then it is clipped to a value within the index range of *seq*.

To sort the text lines in descending order, call the ETAC command **rev_seq** with *seq* as the argument after **fQuickSort** has been called.

Note that the function **fSortSeq** is generally more efficient than this one for an initially nearly sorted sequence.

Examples

The following illustrations show how the **fQuickSort** function can be used.

```
(1) Seq := ["morning", "afternoon", "evening"]; fQuickSort(Seq 1 3);
(2) Seq := ["morning", "afternoon", "evening"]; fQuickSort(Seq 1 |Seq|);
    void rev_seq Seq;
(3) Seq := fReadTextLines("MyTextFile.txt"); fQuickSort(Seq 5 10);
```

Example **(1)** sorts the text elements in the sequence `Seq` in ascending order, resulting in the same sequence having been modified to ‹`["afternoon", "evening", "morning"]`›.

Example **(2)** sorts the text elements in the sequence `Seq` in descending order, resulting in the same sequence having been modified to ‹`["morning", "evening", "afternoon"]`›.

Example **(3)** assumes that the specified file exists, and sorts the order of the text lines at indices `5` to `10` of an internal copy of that file, returning the sorted lines in `Seq`. Note that, in this example, the **`fReadTextLines`** function returns a string sequence containing the data of the specified file.

Other Information

fSortSeq ♦

fReadTextLines

fReadTextLines *file-path* → *str-seq* | ?	@F_READ_TEXT_LINES
file-path	A string stack object.
str-seq	A string sequence.

Details

Reads the text lines of a text file (*file-path*) into a string sequence (*str-seq*).

The text file specified by *file-path* can contain UTF-8, UTF-16, UTF-32, or Windows-1252 text lines delimited by the EOL characters CRLF, CR, or LF. *str-seq* will not include the EOL characters.

If the file cannot be interpreted as a text file, then this function returns a null stack object (`?`). If the file does not exist, or is unreadable, then an *error event* occurs. ♦

fRemQuotes

fRemQuotes *in-str* → *out-str*	@F_REM_QUOTES
in-str	A string stack object.
out-str	A string stack object.

Details

Trims a string (*in-str*) by removing leading and trailing single or double quotes and then spaces, leaving the result (*out-str*) on the object stack.

in-str is a string that may contain leading and\or trailing single-quote (‘`'`’ U+0027) or double-quote (‘`"`’ U+0022) characters. Those characters are removed first, then leading and trailing spaces (U+0020) are removed next.

out-str is the same as *in-str* but with the said characters removed.

Examples

The following illustrations show how the **`fRemQuotes`** function can be used.

```
(1) fRemQuotes("\"text str\"");
(2) fRemQuotes("'\"text str'\"");
(3) fRemQuotes("'  text str' ");
(4) fRemQuotes(" text str ' ");
(5) fRemQuotes(" text str '");
(6) fRemQuotes("' 'text str ' '");
```

In example **(1)**, the input string is ‹`"text str"`›, and the function returns ‹`text str`›.

In example **(2)**, the input string is ‹`'"text str'"`›, and the function returns ‹`text str`›.

In example **(3)**, the input string is ‹`'   text str' `›, and the function returns ‹`text str'`›.

In example **(4)**, the input string is ‹` text str ' `›, and the function returns ‹`text str '`›.

In example **(5)**, the input string is ‹` text str '`›, and the function returns ‹`text str`›.

In example **(6)**, the input string is ‹`' 'text str ' '`›, and the function returns ‹`'text str '`›. ♦

fRepeatStr

fRepeatStr *str len* → *out-str*	@F_REPEAT_STR
str	A string stack object.
len	A non-zero integer stack object.
out-str	A string stack object.

Details

Creates a string consisting of a given string (*str*) repeated a specified number (*len*) of times.

If *len* is zero, *out-str* will be an empty string.

Example

The following illustration shows how the **fRepeatStr** function can be used.

(1) `fRepeatStr("string " 3);`

Example **(1)** returns the string ‹`string string string `›. ♦

fReplSubStr

fReplSubStr *pat repl-str in-str* → *out-str*	@F_REPL_SUB_STR
pat	A string stack object, or a string sequence.
repl-str	A string stack object, or a string sequence.
in-str	A string stack object.
out-str	A string stack object.

Details

Replaces all substrings of a string (*in-str*) that match a *pattern string* or a sequence containing *pattern strings* (*pat*) with a specified string or strings (*repl-str*). Only the parts of *in-str* that match the *zero-blocks* (<0**...**>) of *pat* are replaced.

pat is a *pattern string* or a sequence containing *pattern strings*, but must not contain any <*n*...> *blocks* where *n* is greater than 0. If the strings in *pat* do not contain any *blocks* then those strings are assumed to be contained within a single *zero-block* (ie: <0**...**>). There can be more than one *zero-block* in *pat* but they must not be nested. If *pat* is a sequence, then the first element of that sequence is the main *pattern string*, the other elements are custom *pattern strings* beginning with custom pattern 0. The syntax for the strings in *pat* is as indicated under the heading **Additional Information**, except that those strings can contain only <0**...**> *blocks* (or no *blocks*).

Important Note

If *pat* or any element of it (if *pat* is a sequence) does not have a valid syntax, the consequence is unpredictable. **fReplSubStr** does not cater for patterns with an invalid syntax.

A match occurs when at least one substring of *in-str* matches *pat*. If no match occurs, *out-str* will be the same as *in-str*, otherwise, *out-str* will be *in-str* with the appropriate substrings matching the *zero-blocks* in *pat* replaced with *repl-str*. If *repl-str* is a string then it will replace all matching substrings.

If *repl-str* is a sequence, then the strings in it must correspond to at least the number of matching substrings. Substrings in *in-str* matching the *zero-blocks* in *pat* are replaced with the corresponding strings in *repl-str* in the order in which the substrings occur from left to right.

Examples

The following illustrations show how the **fReplSubStr** function can be used.

```
(1) fReplSubStr("NUM:<0%%`d>" "10" "abcdNUM:24efgNUM:3hijNUM:40");
(2) fReplSubStr("NUM:<0%%`d>" ["7", "66", "204"]
    "abcdNUM:24efgNUM:3hijNUM:40");
(3) fReplSubStr("three" "four" "three people dug three holes");
```

Example **(1)** illustrates how **fReplSubStr** works when *repl-str* is a string ("10"). The first argument, *pat*, is matched by three substrings of the third argument, *in-str*. The three matching substrings are: ‹NUM:24›, ‹NUM:3›, ‹NUM:40›. The number in each of those substrings matches the *zero-block*, <0%%`d>, of *pat*. The second argument, *repl-str*, therefore replaces those numbers. That is to say, the second argument replaces those substrings matching the *zero-block*. In this case, the said numbers are replaced with 10, the second argument. The function, therefore, returns the string ‹abcdNUM:**10**efgNUM:**10**hijNUM:**10**›.

Example **(2)** is the same as example (1), except that the second argument, *repl-str*, is a sequence of three strings that correspond to the three matched numbers mentioned in example (1). Those three numbers, therefore, are replaced by the corresponding three strings in *repl-str*. The function returns the string ‹abcdNUM:**7**efgNUM:**66**hijNUM:**204**›.

Note that the number of elements of *repl-str* must be at least the same as the number of substrings matching the *zero-blocks*, otherwise an ETAC *error event* will occur. In some cases, the number of matching strings may not be known in advanced, so *repl-str* should be a sequence only when that number can be predicted.

The function in example **(3)** returns the string ‹four people dug four holes›, because the first argument, *pat*, is equivalent to ‹<0three>› since it does not contain a *zero-block*.

Additional Information

See **Pattern String Matching** under chapter 3 of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf. ♦

fRunETACFile

fRunETACFile *etac-code* *arg-str* → *rtn-cde*	@F_RUN_ETAC_FILE
etac-code	A string or memory stack object.
arg-str	A string stack object.
rtn-cde	An integer stack object.

Details

Runs an ETAC (or TAC) file (*etac-code*) with an argument string (*arg-str*) as it would be run from RunETAC.exe.

etac-code is either a file path specification to an ETAC code file to run, or a memory stack object containing the ETAC code. The ETAC code is such that it would normally be run from RunETAC.exe, so it therefore expects a string parameter (*arg-str*) on the TAC stack.

arg-str is the string argument for the ETAC code specified by *etac-code*.

If an ETAC *error event* occurs while the ETAC code is executing, this function returns the TAC error code (*rtn-cde*) for that *error event*, otherwise the function returns :#TAC_RTN_SUCCESS:.

If *rtn-cde* is not :#TAC_RTN_SUCCESS:, then the TAC object stack is restored to the same condition that it was before the ETAC code was executed. If *rtn-code* is :#TAC_RTN_SUCCESS:, then the object stack is not restored. This allows the ETAC code to return stack objects on the object stack if so designed.

This function pulls off and saves the dictionaries that are on the TAC dictionary stack, except for a replicate of the "Main" dictionary, before the ETAC code is run. After the ETAC code has completed, the saved dictionaries are restored. It is necessary for the function to save and restore the dictionaries on the dictionary stack to simulate running the ETAC code from RunETAC.exe, otherwise if the current dictionaries are left on the dictionary stack, the executing ETAC code could modify them causing unpredictable behaviour after it has completed. Also, the running ETAC code could itself behave unpredictable if it links into the dictionaries existing before the call.

Examples

The following illustrations show how the **fRunETACFile** function can be used.

```
(1) void fRunETACFile("MyDir\\ShowFile.btac" "InfoFile.txt");
(2) FileData @= &1; FileData += "..."; Rtn := fRunETACFile(FileData "");
(3) Rtn := fRunETACFile((@ &1 + "...") "");
```

Example **(1)** executes ShowFile.btac directly.

Example **(2)** creates a memory stack object, initialises it with *ETAC text script* (indicated by the ellipsis), then executes the *ETAC text script* directly from that memory stack object. The *ETAC text script* is such as would be capable of being run via RunETAC.exe.

Example **(3)** is a more concise way of writing the code in example (2).

Other Information

fExecETACStr ♦

fShowBusy

fShowBusy *show*	@F_SHOW_BUSY
show	An integer stack object containing a logical boolean value.

Details

Shows or hides (*show*) a busy message: "Processing Request". The function shows only a dialog box caption containing the message.

This function can be nested. If *show* is true, then the busy message appears; if *show* is false then an existing busy message is dismissed if the function was called with true the same number of times that it was called with false. ♦

fSortSeq

fSortSeq *in-seq*$_1$ → *in-seq*$_2$	@F_SORT_SEQ
in-seq$_1$	A string sequence.
in-seq$_2$	The modified string sequence *in-seq*$_1$.

Details

Sorts the elements of a string sequence (*in-seq*$_1$) returning the same string sequence (*in-seq*$_2$) with the text lines in ascending order.

This function uses the "insertion sort" algorithm, and is efficient for an initially nearly sorted sequence.

To sort the text lines in descending order, call the ETAC command **rev_seq** with the returned value of this function as the argument.

Examples

The following illustrations show how the **fSortSeq** function can be used.

(1)	`Seq := ["morning", "afternoon", "evening"]; void fSortSeq(Seq);`
(2)	`Seq := ["morning", "afternoon", "evening"]; void rev_seq fSortSeq(Seq);`
(3)	`RtnSeq := fSortSeq(fReadTextLines("MyTextFile.txt"));`

Example **(1)** sorts the text elements in the sequence `Seq` in ascending order, resulting in the same sequence having been modified to ‹`["afternoon", "evening", "morning"]`›.

Example **(2)** sorts the text elements in the sequence `Seq` in descending order, resulting in the same sequence having been modified to ‹`["morning", "evening", "afternoon"]`›.

Example **(3)** assumes that the specified file exists, and sorts the order of the text lines of an internal copy of that file, returning the sorted lines in `RtnSeq`. Note that, in this example, the **fReadTextLines** function returns a string sequence containing the data of the specified file.

Other Information

fQuickSort ♦

fStrInSeq

fStrInSeq *str-seq substr* → *idx*	@F_STR_IN_SEQ
str-seq	A string sequence.
substr	A string stack object.
idx	A non-zero integer object.

Details

Returns the index (*idx*) of the first occurrence of a substring (*substr*) existing in a string sequence (*str-seq*). The search is case-sensitive.

idx is the index of the first string in the sequence *str-seq* that contains the substring *substr*. *idx* is `0` if no substring is found.

Example

The following illustration shows how the **fStrInSeq** function can be used.

(1)	`fStrInSeq(["good morning", "good afternoon", "good evening"] "after");`

Example **(1)** returns the value `2` on the object stack. ♦

fStrToLines

fStrToLines *str eolchrs* → *str-seq*	@F_STR_TO_LINES
str	A string stack object.
eolchrs	A string stack object.
str-seq	A string sequence.

Details

Converts from a string (*str*) containing text lines separated by EOL (end-of-line) characters (*eolchrs*) to a sequence of text lines (*str-seq*) without the EOLs.

str is a string that typically contains text lines separated by the string in *eolchrs*.

eolchrs is a *pattern string* that indicates how the text lines within *str* are separated. For example, the string `"line 1\r\nline 2\r\nline 3"` (ie: ‹`line 1`CRLF`line 2`CRLF`line 3`›) contains three text lines if *eolchrs* is `"\r\n"` (ie: CRLF). Note that *eolchrs* can be ‹`"[{\r\n}\r\n]"`› which checks for CRLF, CR, and LF EOL characters.

str-seq is a sequence containing the separate text lines within *str*, but without the EOL characters. If *str* is an empty string, then *str-seq* will be an empty sequence. If *eolchrs* is an empty string, then *str-seq* will contain the single element *str* if *str* is not an empty string.

This function is typically used to extract a sequence of text lines from a text file, as in the following example:

```
fStrToLines(mem_to_str read_file "MyTextFile.txt" "\r\n");
```

Examples

The following illustrations show how the **fStrToLines** function can be used.

(1) `fStrToLines("line 1line 2line 3" "");`
(2) `fStrToLines("line 1line 2line 3" "\r\n");`
(3) `fStrToLines("" "");`
(4) `fStrToLines("" "\r\n");`
(5) `fStrToLines("line 1\r\nline 2\r\nline 3" "\r\n");`
(6) `fStrToLines("line 1:line 2:line 3:" ":");`
(7) `fStrToLines("line 1\rline 2\r\nline 3" "[{\r\n}\r\n]");`

Examples **(1)** and **(2)** return a sequence with a single string which is the same as the first argument to the function (ie: ‹`["line 1line 2line 3"]`›).

Examples **(3)** and **(4)** return an empty sequence because the first argument is an empty string.

Examples **(5)** to **(7)** return the sequence ‹`["line 1", "line 2", "line 3"]`›.

Other Information

fLinesToStr ▪ fReadTextLines ♦

fToBinStr

fToBinStr *num size* → *str*	@F_TO_BIN_STR
num	An integer stack object.
size	An integer stack object.
str	A string stack object.

Details

Converts an integer (*num*) to a binary string (*str*) of specified size (*size*). The binary string will consist of 0's and 1's.

num is the integer desired to be represented as a two's complement binary string. *num* may be a negative integer.

size is the desired number of binary digit characters to be contained in *str*. If *size* is smaller than the total number of binary digits, then the most significant binary digit characters beyond *size* digits will be truncated. If *size* is greater than the total number of binary digits, then the extra digit characters will be leading zeros. If size is *zero* then *str* will be an empty string. If *size* is negative, then *str* will be the full conversion of *num* without leading zeros.

str is a string containing a series of binary digit characters, '0' and '1'. *str* will represent the binary equivalent of *num* with *size* least significant binary digits.

Examples

The following illustrations show how the **fToBinStr** function can be used.

```
(1) fToBinStr(25 5);
(2) fToBinStr(25 -1);
(3) fToBinStr(25 2);
(4) fToBinStr(25 10);
(5) fToBinStr(-1 -1);
```

Examples **(1)** and **(2)** return the string ‹11001›. This is the full conversion of 25 to binary.

Example **(3)** returns the string ‹01›. Notice that the string contains only 2 least significant digits of ‹11001›.

Example **(4)** returns the string ‹0000011001›. The leading zeros may be truncated if desired via the ETAC command **trim_str**.

Example **(5)** returns the string ‹11111111111111111111111111111111› (32 ones). ♦

fToBoolStr

fToBoolStr *bool* → *str*	@F_TO_BOOL_STR
bool	An integer stack object.
str	A string stack object.

Details

Converts a boolean value (*bool*) to a string (*str*) representing that value.

bool is an integer where non-zero represents true, and zero represents false.

str will contain the text ‹true› if *bool* is non-zero, otherwise it will contain the text ‹false›. ♦

fToHexStr

fToHexStr *num size* → *str*	@F_TO_HEX_STR
num	An integer stack object.
size	An integer stack object.
str	A string stack object.

Details

Converts an integer (*num*) to a hexadecimal string of specified size (*size*).

num is the integer desired to be represented as a two's complement hexadecimal string. *num* may be a negative integer.

size is the desired number of hexadecimal digit characters to be contained in *str*. If *size* is smaller than the total number of hexadecimal digits, then the most significant hexadecimal digit characters beyond *size* digits will be truncated. If *size* is greater than the total number of hexadecimal digits, then the extra digit characters will be leading zeros. If size is *zero* then *str* will be an empty string. If *size* is negative, then *str* will be the full conversion of *num* without leading zeros.

str is a string containing a series of hexadecimal digit characters, '0' to '9' and 'A' to 'F' (note that the hexadecimal characters 'a' to 'f' will be in uppercase). *str* will represent the hexadecimal equivalent of *num* with *size* least significant hexadecimal digits.

Examples

The following illustrations show how the **fToHexStr** function can be used.

```
(1) fToHexStr(251384 5);
(2) fToHexStr(251384 -1);
(3) fToHexStr(251384 2);
(4) fToHexStr(251384 10);
(5) fToHexStr(-1 4);
```

Examples **(1)** and **(2)** return the string ‹3D5F8›. This is the full conversion of 251384 to hexadecimal.

Example **(3)** returns the string ‹F8›. Notice that the string contains only 2 least significant digits of 3D5F8.

Example **(4)** returns the string ‹000003D5F8›. The leading zeros may be truncated if desired via the ETAC command **trim_str**.

Example **(5)** returns the string ‹FFFF›. ♦

fToWinCC

fToWinCC *str* → *wcc* | ?	@F_TO_WIN_CC
str	A string stack object.
wcc	An integer stack object.

Details

Converts the first *w-char* character in a string (*str*) to its Windows-1252 character code (*wcc*). Returns the character code as an integer.

The first character of *str* must be a UCS-2 (BMP Unicode scalar value) character, otherwise the consequence is undefined. An empty string for *str* will result in zero being returned. If the conversion cannot be made, then a null stack object (?) will be returned.

wcc will be less than 256.

Examples
The following illustrations show how the **fToWinCC** function can be used.

```
(1) fToWinCC("");
(2) fToWinCC("hello");
(3) fToWinCC("‡"); [* Unicode U+2021. *]
(4) fToWinCC("?");
(5) fGetStrU("אalef"); [* First character: Unicode U+05D0. *]
```

Example **(1)** returns 0.

Example **(2)** returns 104 (for ‘h’).

Example **(3)** returns 135 (for Windows-1252 ‘‡’).

Example **(4)** returns 63 (for ‘?’).

Example **(5)** returns a null stack object (?) because the first character of the input string is not a member of the Windows-1252 character set. ♦

fTrimQuotes

fTrimQuotes *str* → *out-str*	@F_TRIM_QUOTES
str	A string stack object.
out-str	A string stack object.

Details
Removes single and double quotes from only the ends of a string (*str*).

out-str is the same as *str* but with the outer single (‘'’ U+0027) or double (‘"’ U+0022) quote characters removed. If *str* does not contain the same single or double quotes at both ends, then *out-str* will be the same as *str*.

Examples
The following illustrations show how the **fTrimQuotes** function can be used.

```
(1) fTrimQuotes("");
(2) fTrimQuotes("hello");
(3) fTrimQuotes("' hello'");
(4) fTrimQuotes(' "hello"');
(5) fTrimQuotes('"');
(6) fTrimQuotes("\"hello'");
```

Example **(1)** returns an empty string.

Example **(2)** returns the string ‹hello›.

Example **(3)** returns the string ‹ hello›.

Example **(4)** returns the string ‹ "hello"›.

Example **(5)** returns the string ‹"›.

Example **(6)** returns the string ‹`"hello'`› because the end quotes are not the same. ♦

fTrimStrWS

fTrimStrWS *str* → *out-str*	@F_TRIM_STR_WS
str	A string stack object.
out-str	A string stack object.

Details

Trims a string (*str*) by removing leading and trailing white-spaces, and replacing inner EOL and TAB character sequences with a single space.

str is the string to be modified.

out-str will be the same as *str* but with leading and trailing whitespaces (WS) removed, and inner TAB (9_{16}) and EOL (CR and LF) character sequences replaced with a single space.

Example

The following illustration shows how the **fTrimStrWS** function can be used.

```
(1) fTrimStrWS("\n\t \v\rThis string\v\r\r\n contains\t\n\fwhitespaces \f\r");
```

Example **(1)** returns the equivalent of the string ‹`This string\vSP contains`SP`\fwhitespaces`›. Within the string, both ‹`\r\r\n`› and ‹`\t\n`› have each been replaced by a single space (shown as SP). ♦

fWriteFile

fWriteFile *file-path* *file-data* *backup* → *str*	@F_WRITE_FILE
file-path	A string stack object.
file-data	A memory stack object.
backup	An integer stack object containing a logical boolean value.
str	A string stack object.

Details

Writes data (*file-data*) to the specified file (*file-path*) after creating a backup if specified (*backup*). No action occurs other than an error message in *str* if the file is not writable.

If *file-data* contains Unicode text whose characters are all a subset of the Windows-1252 character set, then the written file will be a Windows-1252 file.

If *backup* is `true`, and the file to be written already exists on disk, a backup of that disk file is made before the specified file is written. If *file.ext* is the format of file name specified in *file-path*, then the backup file name will be *file*~backup.*ext*. If the backup file already exists then it will be overwritten automatically without warning.

If the specified file could not be written to, a temporary file (of the form ‹TMP*xxxxx*.tmp› where *x* is a digit) containing *file-data* may remain.

str will be an empty string if successful, otherwise it will be an error message.

If *file-data* is required to be written in a data form other than the one specified internally, then the following technique can be used. In the illustration below, if the characters in `FileData` are not all a subset of the Windows-1252 character set, the data is temporarily converted to UTF-8 with a BOM before being written; the original data in `FileData` remains unmodified.

```
Msg := fWriteFile(Path to_utf8 :!MO_BOM: FileData Backup);
```

In the example above, **to_utf8** is an ETAC command that converts a memory object (or a string) to the UTF-8 format if possible. :!MO_BOM: writes the data with a BOM signature (if a BOM signature is not desired, :!MO_BOM: is omitted). Other similar commands to **to_utf8** are **to_utf16** and **to_utf32**. Note that if all the characters in FileData are a subset of the Windows-1252 character set, the data is written as a Windows-1252 file (single-byte characters) rather than as a UTF-8 file. If the data in FileData cannot be converted to UTF-8 format, then the data is written as given (without a BOM signature). ♦

Bibliography

The Official ETAC Programming Language copyright © Victor Vella (2020).

Glossary

D

data object

The container of an ETAC dictionary used as a programmer-defined data structure consisting of stack objects identified by name. The dictionary itself is identified by the name defined by the private pre-processor definition ‹_DATA_DICT›.

E

error event

The situation that occurs when the action of an active stack object can no longer proceed. In such a case, the ETAC interpreter intercepts the action and takes appropriate action which typically consists of ending the *main ETAC session*, unless the *error event* is trapped by appropriate ETAC code.

ESL function identifier

The pre-processor definition name identifying an *ETAC function* within an *ETAC script library*. An *ESL function identifier* is specified by the programmer to include the corresponding *ETAC function* definition into *ETAC text script*.

ETAC function

The container of a special ETAC created procedure that creates a local dictionary then assigns the object stack arguments to that dictionary before calling the programmer-defined procedure.

ETAC script library

An *ETAC text script* file containing *ETAC function* definitions designed in such a way that the same required function is allocated only once within a *main ETAC session*.

ETAC session

The period devoted to the processing of ETAC code by the TAC processor after having been processed by the interpretation part of the ETAC interpreter. New *ETAC sessions* can exist within a given *ETAC session* for different ETAC code. Therefore, a given *ETAC session* can produce a new *ETAC session* (relating to different ETAC code from the given *ETAC session*) so that when the new *ETAC session* ends, the given *ETAC session* resumes.

ETAC text script

ETAC program code that is in human readable and writable text form. A file containing only *ETAC text script* typically has an extension of ‘etac’. Note that the term “ETAC text script” is used in the same sense as the word “code”, as in “ETAC text script code”.

M

main ETAC session

An *ETAC session* and all other new *ETAC sessions* produced directly or indirectly from that *ETAC session*, but not itself produced from any other *ETAC session*. A *main ETAC session* is typically begun via the RunETAC.exe and the AppETAC.dll computer programs.

P

pattern string

A *pattern string* is a string which is composed of characters to be matched literally and special characters that indicate predefined patterns to be matched or used to indicated how the pattern matching process is to be performed. *Pattern strings* are unique to, and defined by, the ETAC™ programming language. *Pattern strings* are analogous to "regular expressions" in other programming languages.

See **Pattern String Matching** under chapter 3 of the "*The Official ETAC Programming Language*" document, ETACProgLang(Official).pdf, for more information including the syntax of a *pattern string*.

U

u-char

A Unicode scalar value. A *u-char* is equivalent to a UTF-32 code unit. The size of a *u-char* in an ETAC string is two or four bytes (one or two *w-chars*, respectively). However, a *u-char* size as a character is considered to be one unit in length. Note that a surrogate pair is one *u-char* (even though it is two *w-chars*). A surrogate code point is <u>not</u> a *u-char* (it is a *w-char*).

W

w-char

A Unicode code point in the BMP (Basic Multilingual Plane). A *w-char* is equivalent to a UTF-16 code unit. The size of a *w-char* in an ETAC string is two bytes. However, a *w-char* size as a character is considered to be one unit in length. Note that a surrogate code point is one *w-char*.