SeeBeyond ICAN Suite

Monk Developer's Reference

Release 5.0.5 for Schema Run-time Environment (SRE)

SEEBEYOND®

Version 20050406090124.

Contents

Chapter 11

Conversion Procedures 244

Chapter 12

File I/O Expressions 252

Chapter 13

System Interface Functions 288

Chapter 14

Standard Procedures 300

Chapter 15

Event Definitions 319

Chapter 19

Math-Precision Functions 390

Chapter 20

Monk Library Functions 416

Chapter 22

Chapter 1

Introduction

This chapter introduces you to this guide, its general purpose and scope, and its organization. It also provides sources of related documentation and information.

1.1 Document Purpose and Scope

This guide is a reference for how to use the SeeBeyond Technology Corporation™ (SeeBeyond™) Monk programming language. This guide was developed to provide a single source of information about the core e*Gate Integrator Monk functions.

This is not a "how to program in Monk" guide. Instead, each function available in the general Monk environment is described in its own section as follows:

- Each description tells what the function does, lists the arguments, and tells what the function returns.
- Each section includes a sample of Monk code showing the function in use.

The core Monk functions are those Monk functions made available with the basic e*Gate installation, as opposed to those made available with a specific add-on product such as an e*Way Intelligent Adapter. The Monk functions made available with an add-on product are described in the documentation for that product.

Important: *Any operation explanations given here are generic, for reference purposes only, and do not necessarily address the specifics of setting up and/or operating individual e*Gate systems.*

1.2 Intended Audience

This document was written for experienced programmers writing Collaboration Rules Scripts in Monk. It assumes that the reader has extensive training and/or experience in computer programming skills.

1.3 Organization of Information

This document is organized topically as follows:

- **"Introduction" —** Gives a general preview of this document, its purpose, scope, and organization.
- **"Monk Basics"** — Explains basic information about the Monk language and how it is used.
- **"Control Flow and Boolean Expressions"** — Explains the Monk functions related to controlling the order of statement execution.
- **"Definition, Binding and Assignment"** — Explains the Monk functions that create and manage global variables.
- **"Character Functions"** — Explains the Monk functions related to characters; a character is a fundamental data type containing the representation of a single character within the machine's character set.
- **"String Functions"** — Explains the Monk functions related to character strings.
- **"Numerical Expressions"** — Explains the Monk functions related to Numerical Expressions, that is, expressions used for numerical calculations and conversions.
- **"Pairs and Lists"** — Explains the Monk functions related to pairs and lists; a pair is a structured data type having two parts, called the **car** and the **cdr**.
- **"Vector Expressions"** — Explains the Monk functions related to vector expressions; a vector is defined as a series of elements that can be indexed by integers.
- **"Equivalence Testing"** — Explains the Monk functions related to equivalence testing; an *equivalence predicate* is a computational analogue of a mathematical equivalence relation.
- **"Conversion Procedures"** — Explains the Monk functions related to conversion procedures.
- **"File I/O Expressions"** — Explains the Monk functions related to file input and output; Monk supports the ability to open files, read data from files, and write data to files.
- **"System Interface Functions"** — Explains the Monk functions related to System Interface functions. These functions may be used to find out information *about* files that exist on the system, to load files into the Monk engine, or to execute system commands.
- **"Standard Procedures"** — Explains the Monk functions related to standard procedures.
- **"Event Definitions"** — Explains the Monk functions related to Event definitions.
- **"Date and Time"** — Explains the Monk functions related to date and time.
- **"Interface API Functionality"** — Explains the Monk functions related to interface application program interface (API) functionality.
- **"Debug Procedures"** — Explains the Monk functions related to debug procedures

- **"Math-Precision Functions"** — Explains the Monk functions that provide arithmetic operations with a user-definable precision.
- **"Monk Library Functions"** — Explains all the available Monk Library functions.
- **"International Conversion Functions"** — Explains the international character type conversion functions.
- **"e*Gate Extensions to Monk"** — Explains the Monk functions that are specific to e*Gate version 4.1.
- **"Exception Functionality"** — Explains the Monk exception functions.
- **"Exception Codes"** — Explains the Monk exception codes.

After this introductory chapter, Chapter 2 discusses the basic concepts and applications of Monk. Chapters 3 through 21 describe Monk functions. Chapters 22 and 23 list the Monk exception functions, codes, and messages.

Note: *The functions are grouped according to their use in Monk.*

1.4 Writing Conventions

The writing conventions listed in this section are observed throughout this document.

Hypertext Links

When you are using this guide online, cross-references are also hypertext links and appear in **blue text** as shown below. Click the **blue text** to jump to the section.

For information on these and related topics, see **"Parameter, Function, and Command Names" on page 19**.

Command Line

Text to be typed at the command line is displayed in a special font as shown below.

```
java -jar ValidationBuilder.jar
```

Variables within a command line are set in the same font and bold italic as shown below.

```
stcregutil -rh host-name -rs schema-name -un user-name
-up password -ef output-directory
```

Code and Samples

Computer code and samples (including printouts) on a separate line or lines are set in Courier as shown below.

```
Configuration for BOB_Promotion
```

However, when these elements (or portions of them) or variables representing several possible elements appear within ordinary text, they are set in *italics* as shown below.

path and *file-name* are the path and file name specified as arguments to **-fr** in the **stcregutil** command line.

Notes and Cautions

Points of particular interest or significance to the reader are introduced with ***Note***, ***Caution***, or ***Important***, and the text is displayed in *italics*, for example:

Note: *The Actions menu is only available when a Properties window is displayed.*

User Input

The names of items in the user interface such as icons or buttons that you click or select appear in **bold** as shown below.

Click **Apply** to save, or **OK** to save and close.

File Names and Paths

When names of files are given in the text, they appear in **bold** as shown below.

Use a text editor to open the **ValidationBuilder.properties** file.

When file paths and drive designations are used, with or without the file name, they appear in **bold** as shown below.

In the **Open** field, type **D:\setup\setup.exe** where **D:** is your CD-ROM drive.

Parameter, Function, and Command Names

When names of parameters, functions, and commands are given in the body of the text, they appear in **bold** as follows:

The default parameter **localhost** is normally only used for testing.

The Monk function **iq-put** places an Event into an IQ.

You can use the **stccb** utility to start the Control Broker.

1.5 For information on how to use a specific add-on product (for example, an e*Way Intelligent Adapter), see the user's guide for that product.

Additional Sources of Information

- For information on the general e*Gate programming environment see the *e*Gate Integrator System Administration and Operations Guide*.
- For information on specialized Monk functions, see the documentation for the product that makes them available.

  For example, the **db-sql-select** Monk function, used to perform a SQL SELECT statement on an Oracle database from within Monk, is described in the *e*Way Intelligent Adapter for Oracle User's Guide*.
- For brief information about the syntax of a core Monk function similar to what is provided in this guide, see the online help for the Collaboration Rules Editor.

1.6 SeeBeyond Web Site

The SeeBeyond Web site is your best source for up-to-the-minute product news and technical support information. The site's URL is

http://www.SeeBeyond.com

Chapter 2

Monk Basics

This chapter provides a brief, comprehensive introduction to the Monk programming language.

2.1 Overview

Monk is a specialized algorithmic programming language developed by SeeBeyond. Monk is used with many SeeBeyond products to extend basic functionality. This language is an implementation of the Scheme programming language. Monk has several desirable features that make it extensible and flexible.

About the Monk Programming Language

Monk has latent data types. This means that the data type of a variable is carried with variable and is not defined in a declaration section as in language like Java or C. This makes Monk code simple to write and keep consistent.

Monk has a simple syntax. Once the syntax is mastered, all of Monk functions are interpreted according to the same simple rules regardless of whether the language capabilities have been extended.

These simplicities permit efficient graphical user interface (GUI) design allowing “drag-and-drop” capability in the programmer interface. For further information on Scheme, refer to this Web site:

http://www.swiss.ai.mit.edu/projects/scheme

Chapter Topics

2.2 Data Types

All variables in Monk are associated with a data type. There is no declaration section in Monk (as there is in languages like C or Java) where a variable is assigned its data type. Rather, the data type is determined by the most recent assignment into that variable. This feature is called *latent data types*.

Monk recognizes the following types of arguments.

string	Data type containing zero or more characters. Indicated by a set double quotation marks. Example: **"this is a string"**
character	Data type containing a single alphanumeric character. Indicated by **#**. Examples: **#\a, #\b, #\9** Non-printing characters are referred to by name. Examples: **#\space**, **#\tab**
integer	Data type containing an integer, that is, a numeric value without a fractional part. Examples: **10**, **35**
uint	Data type containing an unsigned integer. Examples: **10**, **35** but not **-123**
int64	Data type containing a 64-bit integer. Range is platform dependent. Examples: **5**, **5000**, **1099511627776**
uint64	Data type containing a 64-bit unsigned integer. Range is platform dependent. Examples: **5**, **5000**, **1099511627776**
ldouble	Data type containing a double precision numerical value with a fractional part. Number of digits of precision is platform dependent. Example: **10995116.27776**
real number	Data type containing numerical value with a fractional part. The fractional part is separated from the integer part with a decimal point. Examples: **10.5**, **35.**, **0.07**
boolean	Data type containing a value of either true **(#t)** or false **(#f)**.
vector	Structured data type of arbitrary elements permitting direct access to any specific element. Indicated by the expression: **#( )**. Example: **#("AA" 10 "CCC" #\b)** is a four element vector.
pair	Structured data type with two fields called the **car** and the **cdr**. Pairs are created by the **cons** procedure. Example: **(cons 'a 'b)** --> **(a . b)**
list	Structured data type defined recursively as either an empty list or a pair whose **cdr** is a list. Examples: **(a b a c)** or **(a . (b . (a . (c . ()))))**.

procedure	Definable using the **lambda** expression. Example: The lambda expression **(lambda (d) (* d 3.1416))** evaluates to a procedure which takes one argument and returns the value of that argument multiplied by pi.
path	Structured value signifying a location within a parsed message. Indicated by a list of message elements separated by dots. Example: **~input%A0X.PID.first-name**
partial path	Structured value signifying a location which contains sub-nodes within a parsed message. It may be further specified to make it a fully qualified **path**. Example: **~input%A0X.PID**
time	Structured data type for use with time functions.
event_struct	Structured event returned by the **$make-event-map** procedure.
interface object	Structured value returned by the **load interface** routine. The loaded .dll adheres to the use of an interface handle and the interface API functionality.
port	Structures value representing the source or destination of data.

2.3 Latent Data Typing

Monk variables are associated with their data types when data is assigned into the variable. A monk variable may change its data type depending upon the data that was last assigned into the variable. This feature of Monk is called *latent data typing*.

For example, you may see code that looks like this:

```
(define myfileptr 0)
(set! myfileptr (open-input-file "C:\mydatafile.txt"))
```

When the variable **myfileptr** is defined, it is associated with an integer data type, because zero is an integer. However, after the **set!** is executed, the variable **myfileptr** is associated with a port data type because the function **open-input-file** returns a port.

The benefits of latent data types are:

- simplifies syntax
- enhances maintainability of code
- makes expressions more compact

In languages like Java or C, which are statically typed, changing the data type of a variable may be difficult. To change a type you must change the declaration of the variable *and* you must examine each occurrence of the variable to ensure that it's usage is consistent with its new type.

With Monk, there is no declaration section to maintain. Where possible, Monk handles data type conversions automatically. Because of latent data types you do not need to worry about numerical conversions between 32-bit representations and 64-bit representations. For example, the table of data types lists the **int** and **int64** data types.

When an integer result is returned that is too great to be held in 32 bits, the variable receiving the numerical result is automatically convert to **int64**.

2.4 Monk Conventions

Discussions of the Monk conventions are divided into the following subtopics:

Naming Conventions on page 24

Identifiers on page 24

Comments on page 25

Whitespace on page 25

Notations on page 26

Literals on page 26

Variables on page 27

Procedure or Function Calls on page 27

2.4.1 Naming Conventions

The names of procedures that always return a Boolean value usually end with a **?**. Such procedures are called predicates.

The names of procedures that store values into previously allocated locations usually end with a **!**. Such procedures are called mutation procedures. By convention, the value returned by a mutation procedure is the assigned value.

When a procedure takes an object of one type and returns a value of an analogous object of another type, **->** appears in the procedure name. For example, **list->vector** takes a list and returns a vector whose elements are the same as those of the list.

2.4.2 Identifiers

Syntax

```
({initial}{subsequent}*)|{peculiar_identifier}
```

Description

Identifiers are a sequence of letters, digits, or “extended alphabetic characters” used to identify the elements of the Monk language.

Parameters

Name	Description
initial	{letter}|{special_initial}
letter	a-z, A-Z

<table>
<tr><td>special_initial</td><td>! $ % & * / : < = > ? ~ _ ^</td></tr>
<tr><td>subsequent</td><td>{initial}|{digit}|{special_subsequent}</td></tr>
<tr><td>digit</td><td>0-9</td></tr>
<tr><td>special_subsequent</td><td>. + - | [% |] , @</td></tr>
<tr><td>peculiar_identifier</td><td>+ - “...”</td></tr>
</table>

Examples

The following are typical identifiers:

```
johnny
list->vector
v17
or
and
```

2.4.3 Comments

Syntax

```
;comments
```

Description

Comments are text inserted within a Monk program. A comment begins with a semicolon **;** and runs from the semicolon to the end of the line in which the semicolon appears. The comment is invisible to Monk.

Example

```
;SYNOPSIS: Multiplies 10 by 20 and displays
;the result
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;STC
(define x 10)
(define y 20)
(display (* x y))
(newline)
```

Special Note

There is also the **comment** procedure, which is used by the GUI to insert comments into monk code. Comments written in this fashion are displayed by the GUI but have no executable effect.

2.4.4 Whitespace

Whitespace characters are spaces, tabs, and newlines. Whitespace is used for improved readability and as necessary to separate tokens from one another.

A token is an indivisible lexical unit such as an identifier or number. Whitespace may occur between any two tokens, but not within a token. Whitespace between tokens is not significant. Whitespace may occur inside a string where it is significant.

2.4.5 Notations

The following notations are used by Monk:

. + -	These are used in numbers, and may also occur anywhere in an identifier except as the first character. A delimited plus or minus sign by itself is also an identifier. A delimited dot (not occurring within a number or identifier) is used in the notation for pairs, and to indicate a rest-parameter in a formal parameter list. A delimited sequence of three successive dots is also an identifier.
()	Parentheses are used for grouping and to notate lists.
'	A single quote character is used to indicate literal data.
`	The backquote character is used to indicate almost-constant data.
, ,@	The character comma and the sequence comma at-sign are used in conjunction with the backquote.
"	The double quote character is used to delimit strings.
****	Backslash is used in the syntax for character constants and as an escape character within a string of constants.
#	The sharp sign is used for a variety of purposes depending on the character that immediately follows it:
#t #f	Boolean constants.
#	This introduces a character constant,
#(	This introduces a vector. Vectors are terminated by).
#b #o #d #x	These are used in the notation for numbers.

2.4.6 Literals

A literal can be one of the following:

- number (an integer or a real)
- string
- character
- path
- boolean
- '()
- quote (datum)

For example, all of the following are literals

10, 10.5

“This is a string”

#\a

#t

‘(“three” “distinct” “strings”)

(quote “three” “more” “strings”)

2.4.7 Variables

An variable is an identifier that names a storage location. A variable is said to be *unbound* or *bound to a location*. The value stored in the location to which a variable is bound is called the variable’s value.

2.4.8 Procedure or Function Calls

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called and the arguments to be passed to it. The procedure and the operand expressions are evaluated, in unspecified order, and the resulting procedure is passed the resulting arguments. Procedure calls may return a value.

The terms *function* and *procedure* are interchangeable in Monk.

Examples:

```
(newline)
```

The **newline** procedure takes no arguments.

```
(string-append "Begin" "the" "Beguine")
```

The **string-append** procedure permits any number of string arguments. It is called here with three arguments.

2.5 The Use of Characters

The following topics discuss characters and how they are used in Monk:

Entering Interpreted Characters as Literals on page 28

Characters to be Escaped in Monk Expressions on page 28

Representing Control Characters in Monk Expressions on page 28

Representing Octal or Hex Characters as Monk Expressions on page 29

2.5.1 Entering Interpreted Characters as Literals

An interpreted character is any character that is parsed as part of the syntax of an expression. For example, when copying a string with the copy expression

```
(copy "copy this string" ~output%MSG.SE.0 "")
```

The double-quote character **"** is an interpreted character marking the boundaries of the string to be copied. After the initial double-quote, the next double-quote to be found is interpreted as the end of the copy-string.

To include a double-quote in the copy-string, the double-quote must be “escaped”. An interpreted character is escaped by preceding it with the backslash **** character, for example, **\"**. The escaped character is then interpreted as a literal character. So, to copy the string:

```
the word "begin" has 5 letters.
```

The copy string is:

```
(copy "the word \"begin\" has 5 letters." ~output%MSG.SE.0 "")
```

The characters **\"** are referred to as an escape sequence.

2.5.2 Characters to be Escaped in Monk Expressions

Within strings, only the double-quote **"** and the backslash **** characters need to be escaped.

Within regular expressions, the backslash precedes characters to be used as regular expression operators. So, as with strings, the double-quote **"** and the backslash **** characters need to be escaped. However, within a regular expression, three backslashes are required to escape the backslash ****.

2.5.3 Representing Control Characters in Monk Expressions

Use the character sequences shown in the following table to represent control characters in Monk expressions:

To represent a control character:	Use this sequence
Alert or audible bell (Control-G)	\a
Backspace (Control-H)	\b
Form-feed (Control-L)	\f
Newline or linefeed (Control-J)	\n
Carriage return (Control-M)	\r
Horizontal tab (Control-I)	\t
Vertical tab (Control-K)	\v

2.5.4 Representing Octal or Hex Characters as Monk Expressions

Use the character sequences shown in the following table to represent octal or hex characters in Monk expressions:

To represent an octal or hex character:	Use this sequence for its character representation:
Hexadecimal value represented by the hex digits, 0–F	#\x*HH* for example, #\x4B
Octal value	#\o*nnn*, for example, #\o113

In a string, use this sequence:	In a regular expression, use this sequence:
\x*HH* for example, \x4B	\x*HH* for example, \x4B
\o*nnn*, for example, \o113	\o*nnn*, for example, \o113

2.6 Regular Expressions

A regular expression is a pattern that represents a set of matching strings. The function **regex** defines the set of strings that will match.

Regular expressions are constructed with ordinary characters and operators. An ordinary character matches itself only. Operators are used to build more complex statements.

The regular expression instruction can be used with the following functions: **change-pattern**, **not-verify**, **verify**, **regex**.

2.6.1 The Simplest Regular Expression

The simplest regular expression consists of a single character, for example, “a”, an ordinary character which matches itself. A slightly more complex regular expression consists of a string of ordinary characters, for example, “abc”. Each character matches itself, therefore, the regular expression, “abc”, matches with any string, that contains “abc”.

2.6.2 Building Complex Regular Expressions

Complex regular expressions are built from simple regular expressions. Link them together by listing them, one after another; no special punctuation is used.

Note: ***regex*** *does not seek an exact match unless you start the string with* **\^** *and end with* **\$.**

To Construct This Regular Expression	Concatenate This Regular Expression	And This Regular Expression	Possible Matches
"ab"	"a"	"b"	"ab"
"\(a*\)a"	"\(a*\)"	"a"	"aaaaa", "aa", "a"
"\(a*\)\(b*\)"	"\(a*\)"	"\(b*\)"	"ab", "aaab", "abbb", "aaaabbbb", "aa"

2.6.3 Regular Expression Operators

Regular expression operators can be used to construct complex pattern-matching expressions. Samples are shown below:

Operator	Usage
\.	Matches any single character, including a newline (but not null).
*reg-ex**	Matches zero or more occurrences of **reg-exp**. The operator, *****, operates on the regular expression immediately preceding ***.** If this is an ordinary character, that character is the regular expression on which ***** operates.
reg-ex\+	Matches one or more occurrences of **reg-exp**. The operator, **\+**, operates on the regular expression immediately preceding **\+**. If this is an ordinary character, that character is the regular expression on which **\+** operates.
reg-ex\?	Matches zero or one occurrence of **reg-exp**. The operator, **\?**, operates on the regular expression immediately preceding **\?**. If this is an ordinary character, that character is the regular expression on which **\?** operates.
reg-ex\{*count*\}	Specify the required number of matches with an integer enclosed in \{ and \}. **reg-exp** must occur exactly *count* times.
reg-ex\{*min*,\}	Specify the minimum required number of matches. **reg-exp** must occur at least *min* times.
reg-ex\{*min*,*max*\}	Specify the minimum and maximum required number of matches. **reg-exp** must occur at least *min* times, but not more than *max* times.
reg-exp1|*reg-exp2*	Matches either *reg-exp1* or *reg-exp2*. The largest regular expression before or after the operator, **|**, is matched. Use **\(** and **\)** to group the regular expressions to remove ambiguity.

Operator	Usage
\[*list*\]	Enclose a list or a range of characters to be matched within brackets. A hyphen may be used to specify a range of matching characters, for example, **[a-z**\\] specifies the set of all lowercase letters as a match. All characters are ordinary within a list, except:
\[^*list*\]	Enclose a list or a range of characters to be excluded from a match within the opening characters, **[^** and a closing bracket, **]**. Otherwise, the syntax is like the matching list, above.
\[[:*class*:]\] \[^[:*class*:]\]	Within a list, a character class expression matches a single character from a given class. A character class expression has the form: [:class:] where class can be:
\(*reg-exp*\)	Remove ambiguity by grouping sub-expressions in **(** and **)**.
\^*reg-exp*	Matches **reg-exp**, if **reg-exp** appears at the beginning of the string matched against.
reg-exp\$	Matches **reg-exp**, if **reg-exp** appears at the end of the string matched against.
\	Backslash activates certain characters to make them operators: . (period), * , +, ?, {, }, |, [,] , (,),^, $ Backslash declares certain operators as ordinary characters, namely: \ Backslash is also used to introduce octal and hex characters; see **"Representing Octal or Hex Characters as Monk Expressions" on page 29**.

2.6.4 Regular Expression Examples

The following table lists common applications of regular expressions.

Application	Regular Expression
Match alternate strings:	"*string*\\|*string*"
Match specific alternate strings at the end of the string matched against, while any data at the beginning matches:	"\\.*\(*string*\)\\|\\.*\(*string*\)"
Match any data, except an empty string:	"\\.\+"
Match a string, at least one character in length, that contains at least a letter:	"\[a-zA-Z\]\+"
Match a string, at least one character in length, that contains at least a numbers:	"\[0-9\]\+"
Match a string that contains at least one character that is not a number:	"\[^0-9\]\+"

Application	Regular Expression
Match a single character that may be a space or a digit:	“\[0-9 \]”
Match a string that contains at least one character that is not a number:	“\[^0-9\]”
Match leading zeros:	“\^0\+”
Match leading spaces:	“\^ \+”
Match trailing spaces:	“ \+\$”
Match a Social Security Number of the nnn-nn-nnnn:	“\[0-9\]\{3\}-\[0-9\]\{2\}-\[0-9\]\{4\}”
Match a telephone number of the format (nnn)nnn-nnnn:	“(\[0-9\]\{3\})\[0-9\]\{3\}-\[0-9\]\{4\}”
Match any 3-character string beginning with ‘t’ and ending with ‘e’:	“t\.e”
Match any 4-character string beginning with ‘(‘ and ending with ‘)’:	“(\.\.)”
Match itself, that is, the character:	“.”
Match a string beginning with ‘f’ followed by zero or more ‘o’s:	“fo*”
Match any string:	“\[0-9\]*”
Match a string of numbers only:	“\^\[0-9\]*\$”
Match any string comprising zero or more strings of the pattern, ‘abc’:	“\(abc\)*”
Match any string comprising zero or more characters:	“\.*”
Match itself, that is, the character:	“*”
Match any string comprising one or more digits:	“\[0-9\]\+”
Match a string comprising one or more spaces:	“\[\]\+”, “\+”, “\[space]\+”
Match any string comprising one or more strings of the pattern, ‘abc’:	“\(abc\)\+”
Match any string comprising one or more characters:	“\.\+”
Match a string comprising one or more ‘o’s between the characters ‘d’ and ‘g’, for example, ‘dog’, ‘doog’, and ‘dooooog’, but not ‘dg’:	“do\+g”
Match itself, that is, the character:	“+”
Match any string comprising zero or one digits:	“\[0-9\]\?”
Match any string comprising zero or one string of the pattern, ‘abc’:	“\(abc\)\?”
Match any string comprising zero or one character:	“\.\?”
Match a string comprising zero or one ‘o’s between the characters ‘d’ and ‘g’, that is, ‘dog’ or ‘dg’:	“do\?g”
Match itself, that is, the character:	“?”
Match the string ‘aaa’:	“a\{3\}”

<table>
<tr><th>Application</th><th>Regular Expression</th></tr>
<tr><td>Match a telephone number of the format ‘nnn-nnn-nnnn’:</td><td>“\[0-9\]\{3\}-\[0-9\]\{3\}-\[0-9\]\{4\}”</td></tr>
<tr><td>Match a Social Security Number of the format ‘nnn-nn-nnnn’:</td><td>“\[0-9\]\{3\}-\[0-9\]\{2\}-\[0-9\]\{4\}”</td></tr>
<tr><td>Match themselves:</td><td>“{” “}”</td></tr>
<tr><td>Match the string ‘banana’, ‘bananana’, and so on, but not ‘bana’:</td><td>“ba\(na\)\{2,\}”</td></tr>
<tr><td>Match the strings ‘banana’ and ‘bananana’ only:</td><td>“ba\(na\)\{2,3\}”</td></tr>
<tr><td>Match either ‘a’ or ‘b’:</td><td>“a\|b”, “babe”, “abe”, “be”, “apple”</td></tr>
<tr><td>Match ‘hello’ or ‘bye’ or ‘later’:</td><td>“hello\|bye\|later”</td></tr>
<tr><td>Match ‘care’ or ‘core’ or ‘cure’:</td><td>“c\(a\|o\|u\)re”</td></tr>
<tr><td>Match ‘aa’ or ‘ab’ or ‘ba’ or ‘bb’:</td><td>“\(a\|b\)\(a\|b\)”</td></tr>
<tr><td>Match itself, that is, the character:</td><td>“|”</td></tr>
<tr><td>Match either the character ‘x’ or the character ‘y’:</td><td>“\[xy\]”, “xyz”, “xzy”, “xabcy”</td></tr>
<tr><td>Match any single character that is part of the set of all uppercase letters, A through Z; the digits, 0 through 9; or the characters, ‘$’ or ‘!’:</td><td>“\[A-Z0-9$!\]”</td></tr>
<tr><td>Match any single character that is part of the set of the digits, 0 through 9, or the characters ‘[’, ‘]’, and ‘-’:
(Note that to match the close square bracket (]) it must be at the beginning of the list.)</td><td>“\[][0-9-\]”</td></tr>
<tr><td>Match themselves:</td><td>“[” “]”</td></tr>
<tr><td>Match any single character that is not ‘x’ or ‘y’:</td><td>“\[^xy\]”</td></tr>
<tr><td>Match any single character that is not part of the set of all uppercase letters, A through Z; the digits, 0 through 9; or the characters, ‘$’ or ‘!’:</td><td>“\[^A-Z0-9$!\]”</td></tr>
<tr><td>Match any single character that is not part of the set of the digits, 0 through 9, or the characters ‘[’, ‘]’, and ‘-’:</td><td>“\[^0-9[]-\]”</td></tr>
<tr><td>Match a lowercase letter:</td><td>“\[[:lower:]\]”</td></tr>
<tr><td>Match any string of at least one character followed by zero or more white space characters:</td><td>“\.\+\(\[[:blank:]\]\|\[

[:space:]\]\)*”</td></tr>
<tr><td>Enclose a set of alternates:</td><td>“\(anti\|pro\)\.\+tion”</td></tr>
<tr><td>Enclose a complex regular expression to be operated on by a ‘*’, ‘\+’, or ‘\?’:</td><td>“ba\(na\)*”</td></tr>
<tr><td>Enclose sub expressions within a set of alternates:</td><td>“\.*\(CA\)\|\.*\(WA\)”</td></tr>
<tr><td>Match themselves:</td><td>“(” “)”</td></tr>
<tr><td>Match the string, ‘abc’, if it appears at the beginning of the string matched against:</td><td>“\^abc”</td></tr>
<tr><td>Match a string of one or more zeros at the beginning of the string matched against:</td><td>“\^0\+”</td></tr>
</table>

Application	Regular Expression
Match itself, that is, the character:	“ ^ ”
Match the string, ‘abc’, if it appears at the end of the string matched against:	“\(abc\)\$”
Match a string of one or more zeros at the end of the string matched against:	“0\+\$”
Match itself, that is, the character:	“$”
Match the backslash (\) character: or within a list:	“\\\\” “\[\\\]”

2.7 Format Specification

Syntax

```
"%<flag><width>.<precision>[alt format]<C>"
```

Description

Format specification converts arguments from their internal representation to a printable form. The format specification can be used with several of the expressions detailed in this document.

Parameters

Name	Description
<flag>	Formatting option that modifies the <C> conversion character. Multiple flags can be specified. Not all flags can be used with each data type. See **“Examples” on page 36** for a list if flags that can be used with each data type. - Output is left aligned. + A sign (+ or -) always precedes output. space If the first character to be output is not a sign (+ or -), a space character is prefixed. Only one space is allowed in a format specification. 0 Numbers are right-aligned and padded with leading zeros. # Output includes a decimal point.
<width>	Number equal to 1 or greater. The width of the field is determined by the length of the formatted data but cannot be less than <width>. If the formatted data is narrower than <width>, then the result is left padded or right padded with spaces or zeros depending on other flags. **Note:** *The width of a field cannot be greater than 9,999,999 places.*
<precision>	Number equal to zero or greater. Indicates the number of digits to the right of the decimal.If used, must be preceded by a period to distinguish it from <width>.

Name	Description
[alt format]	Only used with **t, T.** When specified, uses the time format defined by the **mk-time** procedure on **"mktime" on page 373**.
<C>	Conversion character indicating output data type. Data types with capital letters attempt to print that element as Monk-readable text. Lowercase data types print in a normal, text-readable format. Not available for E, F, T, or *; reserved for future use. Select one of the following: a, A — Any Monk object. b, B — Binary output of a number. d, D — Decimal output of a number. Integer (positive or negative). e, E — Exponent output of a number. Floating point number formatted with scientific notation [-]n.me+/-xx. f, F — Fixed output of a number. Floating point number formatted with decimal notation [-]n.m where - is output for negative numbers. i, I — Decimal output of a number. n, N — Number of bytes written so far. o, O — Octal output of a number. s, S — String output. t, T — Time output. x, X — hexadecimal output of a number. * — Use next argument as directive information.

A literal string may be included in the format. For example

```
(format "Cherries are %s" "red.")     =>  "Cherries are red."
```

The following table relates conversion characters to the format flags permitted to the conversion character.

Conversion Character	Permitted Format Flags
a, A	-
b, B	0, +, -, space
d, D	+, -, space
e, E	+
f, F	+, .
i, I	+, -
n, N	none
o, O	0, -, +, #, space
s, S	none
t, T	none
x, X	0, -, +, #, space

Conversion Character	Permitted Format Flags
*	none

Examples

```
(format "%b" "33")              =>   "100001"
(format "%-8c" "Tiger")         =>   "Tiger   "
(format "%07o" "33")            =>   "0000041"
```

These examples demonstrate binary conversion, left-justify using the minus character, and padding with zeros.

The following table lists a variety of inputs, formats and the resulting string.

Note: *The double quotes are not part of the result data. They are included to delimit significant spaces.*

Input	Format Instruction	Result
Floating point format examples		
`12.345`	`%9.0f`	`"       12"`
`12.345`	`%9.1f`	`"     12.3"`
`12.345`	`%9.2f`	`"    12.34"`
`12.345`	`%9.3f`	`"   12.345"`
`12.345`	`%9.4f`	`"  12.3450"`
`12.345`	`%8.4f`	`" 12.3450"`
`12.345`	`%7.4f`	`"12.3450"`
`12.345`	`%6.4f`	`"12.3450"`
`12.345`	`%09.0f`	`"000000012"`
`12.345`	`%09.1f`	`"0000012.3"`
`12.345`	`%09.2f`	`"000012.34"`
`12.345`	`%+-09.2f`	`"+12.34   "`
`-12.345`	`%+-09.2f`	`"-12.34   "`
`12.345`	`%+09.2f`	`"+00012.34"`
`-12.345`	`%+09.2f`	`"-00012.34"`
`12.345`	`%-09.2f`	`"12.34    "`
`-12.345`	`%-09.2f`	`"-12.34   "`
Integer Format Examples		
`123`	`%i`	`"123"`
`123`	`%8i`	`"     123"`
`123`	`%7i`	`"    123"`
`123`	`%-6i`	`"123   "`
`123`	`%-5i`	`"123  "`
`123`	`%+4i`	`"+123"`
`123`	`%+3i`	`"+123"`
Octal Format Examples		

Input	Format Instruction	Result
33	`%o`	`"41"`
33	`% o`	`" 41"`
33	`%09o`	`"000000041"`
33	`%08o`	`"00000041"`
33	`%8o`	`"      41"`
33	`%7o`	`"     41"`
33	`%6o`	`"    41"`
33	`%5o`	`"   41"`
33	`%-9o`	`"41       "`
33	`%+09o`	`"+00000041"`
-33	`%+09o`	`"-00000041"`
33	`%+9o`	`"      +41"`
-33	`%+9o`	`"      -41"`
-33	`%#9o`	`"      -41"`

2.8 Monk and Event Definitions

Creating event definitions to process event data is the fundamental usage of Monk. When you create an event definition, you define how the event is to be parsed into logical hierarchies. You also assign names to those logical units so that data can be accessed more easily. This makes the task of accessing and manipulating the data more straightforward.

The process of mapping event data to a structured event is an implicit verification of the data against the structure. If the elements specified in the event definition don't match the event data, mapping fails. When the event data does map successfully to the event definition, the result is a parsed and labeled a *structured event*.

2.8.1 Contents of an Event Definition

An event definition is the skeleton or blueprint of event data. The event definition describes how to locate data in an event. It is constructed using:

1. **A list of delimiters.** The delimiter list assists in describing the event structures's physical hierarchy and, thereby, how data is to be parsed into its units, from its highest to its lowest level.
2. **A list of nodes.** The node list describes the event's logical structure. You define the logical structure by establishing the criteria by which the physical structure is to be organized. Concurrently, you assign names to your organization, thus enabling clear access to the data components for manipulation. When defining the logical structure, you identify and name:
   - **Ordered groups**—structured event elements that comprise an ordered set (that is, the data elements must exist in the specified order).
   - **Unordered groups**—structured event elements that comprise an unordered set (that is, the data elements can exist in any order).

- **Repetitions**—structured event elements that repeat.
- **Hierarchy**—the event element levels.
- **Constants**—structured event elements that are required.
- **Optionals**—structured event elements that are optional.
- **Fixed-length fields**—structured event elements that have a fixed length.

When the delimiter list and the list of nodes are combined, they form a *structured definition.*

2.8.2 Structured Events

A structured event is created when event data has been mapped to an event definition. You can also think of it as parsed event data. A structured event is the result of the delimiter list and the **$make-event-map** expression. You access data in a structured event using the labels you assign in the node list. The labels represent logical hierarchies and locations for data access.

Following is an example of a structured event. The structured event is created using the delimiter information, the event definition, and the mapped data shown below.

```
"This is an event, and a string.
Delimiters are spaces, commas, and
periods."
```

In this example, the delimiter list specifies that a period (**.**) delimits top-level structured event elements, a comma (**,**) delimits second-level structured event elements, and a space ( ) delimits third-level structured event elements.

Also, the node list specifies that the event is to be labeled "Event." The event will contain one or more top-level structured event elements, to be labeled "Sentence." Sentences will contain zero or more second-level structured event elements, to be labeled "Phrase." Phrases will contain zero or more third-level structured event elements, to be labeled "Word."

Once the structured event is created, you can use the labels from the node list to access event data, as shown in the table below.

Use this label:	To access this part of the event:
Event	This is an event, and a string. Delimiters are spaces, commas, and periods.
Event.Sentence[0]	This is an event, and a string
Event.Sentence[1]	Delimiters are spaces, commas, and periods
Event.Sentence[0].Phrase[0]	This is an event
Event.Sentence[0].Phrase[1]	and a string
Event.Sentence[1].Phrase[0]	Delimiters are spaces
Event.Sentence[1].Phrase[1]	commas
Event.Sentence[1].Phrase[2]	and periods

Use this label:	To access this part of the event:
Event.Sentence[0].Phrase[0].Word[0]	This
Event.Sentence[0].Phrase[0].Word[1]	is
Event.Sentence[0].Phrase[0].Word[2]	an
Event.Sentence[0].Phrase[0].Word[3]	event
Event.Sentence[0].Phrase[1].Word[0]	and
Event.Sentence[0].Phrase[1].Word[1]	a
Event.Sentence[0].Phrase[1].Word[2]	string
Event.Sentence[1].Phrase[0].Word[0]	Delimiters
Event.Sentence[1].Phrase[0].Word[1]	are
Event.Sentence[1].Phrase[0].Word[2]	spaces
Event.Sentence[1].Phrase[1].Word[0]	commas
Event.Sentence[1].Phrase[2].Word[0]	and
Event.Sentence[1].Phrase[2].Word[1]	periods

2.8.3 How Monk Uses Paths to Access Structured Events

A path specifies a structured event location to access. You can use a path in any Monk expression that operates on a structured event. When you access data via a path, you are working with a *copy* of the node data (as a string).

There are two ways to specify paths in Monk expressions. You can specify a complete path or you can specify a partial path.

Complete Path

This path expression represents a *complete path* to data of a structured event. It begins with a tilde (~) and includes the name of the structured event followed by a percent sign (%) and the path elements.

```
~event-name%path_elements
```

Partial Path

This path expression represents a *partial path* to data of a structured event. It begins with a percent sign (%) and includes the path elements.

```
%path_elements
```

Parameters

Name	Description
event-name	The name of the structured event. Optional. If event-name is not specified, the expression represents a partial path.

Name	Description
path_elements	A list of event locations separated by dots (.). Each element can be either: A variable that contains a partial path, number, or node name. A name assigned in the node list to a structured event element or set of structured event elements. An **integer** that represents the structured event element's child position. The first structured event element at a given level is counted as 0.

Data extracted from a structured event is a string. If a path accesses a structured event element and that element is not present in the structured event, the result is an empty string.

The **copy** expression appends data to the end of existing data at a structured event location. This is useful for building strings within a restricted data field.

Data is not appended to the end of an event location if you specify a byte offset. Specifying a byte offset turns off the auto-append feature and overwrites any data that exists in the specified byte locations.

Appended data is truncated if it exceeds the maximum byte length of an event definition. This feature can be used to build strings within a field, for example.

If an expression attempts to place data to a node repetition that exceeds the specified maximum repetition count, a warning is generated and the excessive repetitions are not written to the structured event.

If the path specified has no corresponding location in the structured event definition, an exception is generated.

Delimiter List

Syntax

```
((delimiterspec1)(delimiterspec2)...(delimiterspecN))
```

where the syntax of *delimiterspec* is:

```
delimiter   [delim_type]
```

Description

Elements from the delimiter list are used by the **$make-event-map** expression to specify the event separators.

Delimiters describe the event's physical hierarchy and, thereby, how it is to be parsed into its units, from its highest to its lowest level. List delimiters in their hierarchical order, from highest to lowest.

Parameters

<table>
<tr><th>Name</th><th colspan="2">Description</th></tr>
<tr><td>delimiter</td><td colspan="2">A delimiter that can be represented:
As a string, such as "|" (vertical bar). It must have a length of 1 or more.
As an integer, which is the byte location of the delimiter in the event (if delimiters are declared in the event in a standard location). A length of 1 is assumed.
In the Monk notation for character constants, for example, \#newline.
In the following syntax to represent the byte location and length of the delimiter in the event: (byte_location length)
In the following syntax to represent the beginning delimiter and ending delimiter in the event: ("begin_delim" "end_delim")</td></tr>
<tr><td rowspan="8">delim_type</td><td colspan="2">Type of delimiter. Keywords are:</td></tr>
<tr><td>endofrec</td><td>Delimiter always ends a event element at this level. For example, segments always terminate with \r. This is optional on input mapping, but generated as part of output.</td></tr>
<tr><td>array</td><td>Optional delimiter used for array-type nodes. The array delimiter is the repetition field delimiter used in the HL7 event format for repeating fields.</td></tr>
<tr><td>anchored</td><td>If a delimiter is marked as anchored, the Monk parser looks for that delimiter (begin or end) at the current byte location of the event data.</td></tr>
<tr><td>beginanchored</td><td>If a delimiter is marked, the begin delimiter must occur at the current location of the event data.</td></tr>
<tr><td>endanchored</td><td>If a delimiter is marked as endanchored, the end delimiter must occur at the current location of the event data.</td></tr>
<tr><td>required</td><td>Used only for end delimiters. The delimiter must exist in the data.</td></tr>
<tr><td>separator</td><td>Used for backward compatibility only.</td></tr>
</table>

Examples

In the following list the delimiters are expressed as strings.

```
 ;;; Delimiter List
(define RAS-delm '(
 ("\r" endofrec)
 ("|")
 ("~" array)
 ("^")
 ("&")
 ))
```

As background, the delimiters are declared in the MSH segment as shown below. The byte count appears beneath the MSH segment-id and delimiters:

```
MSH|^~\&
```

In the following list the delimiters are expressed as byte locations.

```
;;; Delimiter List
(define RAS-delm '(
 ("\r" endofrec)
 (3)
 (5 array)
 (4)
 (7)
 ))
;;; Delimiter List
(define RAS-delm '(
 ("~")
 ("*")
 ))
```

In the following list the delimiters are expressed as strings.

In the following list the delimiters are expressed as character constants.

```
;;; Delimiter List
(define RAS-delm '(
 (#\~)
 (#\*)
 )).
```

Node List

Syntax

```
([modifier-list] name-of-node node-type min-rep max-rep "tag"
"default-data" offset length expression1 expression2... expressionN)
```

Description

An argument to the **$make-event-map** expression. Use the node list to define the logical structure of the event.

Notes

If no attributes are set then it uses the attributes from the default delimiter list.

If anchored or beginanchored is specified and no beginning delimiter is specified, then the begin delimiter is inherited from the default delimiter list.

If no end delimiter is specified, an end delimiter is inherited from the default delimiter list.

If an begin delimiter is specified and no end delimiter is given, then an end delimiter is inherited and the required attribute is set.

No other modifiers are inherited. If you set any modifier attribute, then all other attributes from the default delimiter list are cleared.

Examples

```
default delim list:
  (("[" "]") endanchored)
  (("<rep>" "</rep>") array)
  ("+" )

node level 1:
  (Ed)
  Begin Delim: none
  End Delim  : "]"
  attributes : none

  Rep delims:
    Begin Delim: none
    End Delim  : none
    attributes : none

  (Bd)
  Begin Delim: "["
  End Delim  : "]"
  attributes :  required

  Rep delims:
    Begin Delim: none
    End Delim  : none
    attribs    : none

  ((Ed "foo") Ed)
  Begin Delim: none
  End Delim  : "]"
  attributes :  none

  Rep delims:
    Begin Delim: none
    End Delim  : none
```

```
    attribs     : none

  ((Ri Ed) Ed endanchored)
  Begin Delim: none
  End Delim  : "]"
  attributes :  endanchored

  Rep delims:
    Begin Delim: none
    End Delim  : "</rep">
    attribs     : none

  ((Ri (Ed ")"))
  Begin Delim:  none
  End Delim  :  "]"
  attributes :  none

  Rep delims:
    Begin Delim: none
    End Delim  : ")"
    attribs     : none
```

Attributes

Table 1 Attributes of the Node List

Name	Description	
modifier-list	Optional list of modifiers.	
	Bd	Begin delimiter.
	(Bd *delim-type*)	
	BdB	Begin delimiter bind. Designates that if you have a begin delimiter, you must have a matching end delimiter from the same pair.
	Co	Consumer node. If you have written a Monk function to map the data, you must return a length of how much of the data you expect to consume.
	Ed	End delimiter.
	(Ed *delim-type*)	
	EscD	Treats the string as a literal.
	Ex	Specifies that you cannot expand the data map. Only the mapped data will be used.
	ExF	Specifies that you cannot expand the data map. If the data exceeds the map, it will fail and not map any of the data.
	Get	Specifies that you can only *get* data from this node.
	Gr	Group repetitions. Groups disjoint repetitions of a child.
	NofN	Minimum number of occurrences of *N* optional children nodes.

Table 1 Attributes of the Node List (Continued)

Name	Description	
	Nt	Not tagged. Results all characters that are not designated as tagged.
	Pp	Parent precedence. The parent delimiter will take precedence over the child-node delimiters.
	Put	Specifies that you can only *put* data into this node.
	Ri	Array repetition information.
	Sc	Scavenger. Designated characters in the string are consumed before attempting to map the node.
	ScN	Scavenger node. Specifies that the first character in the output node will not be output.
name-of-node	The name you give to the node. Node name limitations are detailed in **"Rules for Naming Nodes" on page 48**.	
node-type	The type of node.	
	ON	Delimited node.
	AN	Any-ordered delimited node. The nodes below an any-ordered node can appear in any order.
	OF	Fixed node.
	AF	Any-ordered fixed node.
	OS	Ordered set. A set represents a group of nodes at the same level. Sets are used to represent a pattern of repeating elements. The nodes below an ordered set must occur in the order specified.
	AS	Any-ordered set. A set represents a group of nodes at the same level. Sets are used to represent a pattern of repeating elements. The nodes below an any-ordered set may occur in an order different from the specified order. (Use this option with care: the event-parsing process can take much longer when this option is specified.)
	ONA	Ordered delimited node-array. A node-array is similar to a set, but the group it represents comprises sub-nodes, instead of nodes at the same delimiter level. Use a node-array to represent a repeating field (where repetitions are delimited by the repetition field array delimiter (for example ~ character). The nodes below an ordered node-array occur in the event in the order specified.

Table 1 Attributes of the Node List (Continued)

Name	Description	
	ANA	Any-ordered delimited node-array. A node-array is similar to a set, but the group it represents comprises sub-nodes, instead of nodes at the same delimiter level. Use a node-array to represent a repeating field (where repetitions are delimited by the repetition field array delimiter (for example ~ character). The nodes below an any-ordered node-array may occur in an order different from the specified order. (Use this option with care: the event-parsing process can take much longer when this option is specified.)
	GTN	Global (external file) template, delimited node. The template Is defined in a file other than the current file.
	LTN	Local template, delimited node. The template is defined in the current file.
	GTF	Global (external file) template, fixed node. The template is defined in the current file.
	LTF	Local template, fixed node. The template is defined in the current file.
	GTS	Global (external file) template, set. The template is defined in the current file.
	LTS	Local template, set.
min-rep	The minimum number of repetitions of the node that must occur when mapping the structured event. Number must be positive. Samples are shown below.	
	min/max	
	1 1	Minimum of one, maximum of one. Non-repeating, required.
	0 1	Maximum of one. Optional.
	0 INF	No maximum. Optional.
	1 INF	Minimum of one, no maximum.
	1 5	Minimum of one, maximum of five.
	5 5	Minimum of five, maximum of five. The event must contain exactly five instances of the element or group to match the event definition.
max-rep	The maximum number of repetitions of the node; no more than this number can occur when mapping the structured event. Number must be positive. Samples are shown in the description of min-rep.	

Table 1 Attributes of the Node List (Continued)

Name	Description
tag	A string that is compared to the node data. If the comparison fails, the map of that node data fails. If unspecified, it defaults to und (undefined). When the node-type is an external template, the *tag* argument is overloaded with the template filename.
default-data	A string to represent the data of the node if node is required and no data has been written to it. If unspecified, it defaults to und (undefined). When the node-type is a template, the *default-data* argument is overloaded with the template symbol.
offset	Number of bytes to count from the first byte (byte 0) of the parent node. If unspecified, it defaults to und (undefined). (In this case, the current node starts at the end of the previous node.) Byte offset is supported for fixed (F) nodes only.
length	Total bytes of data that represent the node. If unspecified, it defaults to und (undefined), meaning the rest of the data or bound by the size of the parsed children. Supported for fixed (F) nodes only. Optionally, you can specify (*start end*) instead of length. Start is the first byte to read and end is the last byte to read (counting from byte 0).
expression	Remainder of list specifying children.

Node Properties Effect

The following table summarizes how node properties affect placement of data into a structured event.

Property	Node Type	Source Data Mapping	Placing Data
min-rep, max-rep	all	Data must contain at least the minimum number of repetitions specified and at most the maximum number of repetitions specified to successfully map.	If an expression attempts to place data in a node repetition that exceeds the specified maximum, a warning is written to the current-warning-port and the process terminates with no action taken.
tag	delimited, delimited-array	Node contents must match the tag or the map fails, that is, represented by regular expression, “\^tag\$”	No impact.
	fixed	Fixes start/location/length of node in data stream.	No impact.
default data	delimited, delimited-array, fixed	No impact.	If no data is placed in a node and it is a required node, then the default data represents the content of the node.

Property	Node Type	Source Data Mapping	Placing Data
length	delimited, delimited-array	No impact.	No impact.
	fixed, byte length declared	Data available for this node must be this length or map fails (if you have optional, trailing fixed nodes in definition, do not declare length in root). If a negative number is specified, the length is determined from the current position to the current parent end, less the bytes specified.	If data written to node exceeds specified length, the data is truncated and a warning is output.
	fixed, no byte length declared	Defaults to und (undefined) meaning the rest of the data.	No impact.

Rules for Naming Nodes

Adhere to the following rules when naming nodes.

1 The following characters are accepted:

A-Z, a-z	(letters)
0-9	(numbers)
+	(plus-sign)
-	(hyphen)
*	(asterisk)
/	(slash)
=	(equal sign)
!	(exclamation point)
?	(question mark)
$	(dollar sign)
_	(underscore)
&	(ampersand)
^	(caret)

2 The first character *cannot* be:

0-9	(numbers)
+	(plus-sign)
-	(hyphen)

3 Node name interpretation is case sensitive.

4 Each event type definition must be uniquely named.

2.8.4 Behavior of Optional Nodes That Contain No Data

This section discusses how optional nodes are assigned attributes to assist in the data output process. The following table identifies the terms that are necessary for this discussion.

Node Type	Meaning	Description
RNU	required, non-unique	required, untagged
SU	strongly unique	required, tagged
WU	weakly unique	optional, tagged
NU	non-unique	optional, untagged

In the first phase, the event structure is created with the **$make-event-map** procedure and the initial assignment of nodes types is based on the attributes of the node being defined.

During the second phase, attributes are altered based on parent-sibling and sibling-sibling relationships. The following list identifies the possible parent/sibling promotions:

- A strongly unique node promotes its preceding sibling from non-unique to required, non-unique status.
- A strongly unique child node promotes its parent from non-unique to weakly unique status.

The third phase of promotion occurs at run time when data is passed into the structured event:

- If a node is non-unique (NU) and has data in any of it's trailing siblings, NU sibling's output data to represent that node is generated.
- If the above condition is not fulfilled, an output node is generated only as the result of the sibling to sibling and child to parent interactions.

The table below identifies whether or not a node will be generated after all promotions have taken place.

Node Type	Output Node Generated?
SU	yes
RNU	yes
WU	no
NU and data in sibling	yes
NU and no data in sibling	no

2.8.5 Dynamic Parsing of Data

When adding data to an existing child node, data present in its parent node is marked invalid.

When data is written to a child node that does not exist, the data is parsed from the parent node into the children nodes. Data is added to the child node, and data in the parent node is marked invalid.

When data is added to a parent node, but the parent node does not contain valid data, the following happens:

1. Data is re-constituted from the children nodes.
2. The child subtree is deleted.
3. Data is added to the parent node.

When data is added to a parent node, and the parent node contains valid data, the following happens:

1. The child subtree is deleted.
2. Data is appended to the parent node.

2.8.6 Referencing an Instance of a Repeating Node

To specify an instance of a repeating node, the syntax is:

```
pathelement[index]
```

index

An integer that represents the repetition desired or can be replaced by a variable name, as discussed below.

For example:

```
~input%ROOT.play-it-again-sam[5]
```

the sixth repetition of the structured event element **play-it-again-sam**.

If a repetition is not specified for a structured event element, the first repetition is accessed by default if followed by path elements. For example, the following two paths are equivalent:

```
~input%ROOT.NTE[0].FONE
~input%ROOT.NTE.FONE
```

Referencing Data with Byte Count

Byte positions can be specified as the final path element in the list. Note that specifying byte positions in the path when placing data to a structured event turns off the auto-append feature and overwrites any data that may exist in the specified byte locations.

There are two methods for specifying byte positions. The first method specifies relative addressing while the second specifies absolute addressing. The syntax for these two methods is shown below.

finalpathelement:*byte_offset*,*length*

or

finalpathelement:*byte_offset*- *end_byte*

byte_offset

The beginning byte position, counted from the first byte of the structured event data location (the starting position is inclusive). That is, the first byte is counted as 0.

length

The number of bytes to be accessed. Length is optional. You can leave it out or use the keyword END to indicate "from *byte_offset* to the end of the structured event element."

end_byte

The ending byte position. The ending position is exclusive, the up to end_byte is absolute, and an *end_byte* is optional. You can leave it out or use the keyword END to indicate "from *byte_offset* to the end of the event element."

For example, the following path elements access eight bytes, starting at the third byte (byte 2) and ending at the tenth byte (byte 9) of the N1 event location.

```
N1:2,8
N1:2-10
```

The path elements below are also equivalent. They each access from the third byte (byte 2) to the end of the N1 event location.

```
N1:2,
N1:2,END
N1:2-
N1:2-END
```

Length Specification, When Assigning Data to Structured Event

If you use a length specification in the path expression and the data to be assigned is shorter than the length specified, the string is padded with trailing spaces. For example:

```
(copy "AAA" ~output%root.node.field:0,5)
```

copies the string "AAA" to the output location.

In all cases, the assigned data is left-justified in the destination location.

Examples

This path:	Locates this event element:
~input%MSG	The entire structured event data. (A structure's root node represents the complete event.)
~input%MSG.ST	The complete string represented by the ST node, including all repetitions if it is a repeating node, and all children.
~input%MSG.ST[0]	The first repetition of ST if it is a repeating node, or the first child of ST if it has children.

This path:	Locates this event element:
~input%MSG.ST.2	The third field first repetition of the ST segment of the MSG node.
~input%MSG.DTM[2].4	The fifth field of the third repetition of the DTM segment of the MSG node.
~input%MSG.N1[0].5	The sixth field of the first repetition of the N1 node (of the MSG node).
~input%MSG.MIT[4].N1[5].PER[2].6	The seventh field of the third repetition of the PER (of the sixth repetition of the N1 node (of the fifth repetition of the MIT node (of the MSG node))).

2.8.7 Use of Variables to Represent Path Elements

A variable that contains a path, a number, or a symbol can be used in a path. Sample uses include using a variable name for a frequently accessed location, substituting a variable for an instance index in a **do** loop expression, or using variables to reference byte counts.

Variable names within a path are denoted by angle brackets. For example:

```
<var_name>
```

When assigning a path value to a variable, you must precede the path with either a percent sign (%) or a tilde (~). For example:

```
(define ETC %MSH.EVN.1)
```

Examples

This path:	Uses a variable to:
~input%<ETC>	Represent a path. ETC is a variable with the value %MSH.EVN.1 (as defined above).
~input%MSG.DTM[<i>].4	Represent an repetition. This path might be used within a **do** loop expression; the value of< i> would be the current value for the loop's iteration counter.
~input%MSG.CID.19:<i>,<j>	Represent byte offset <i> and length <j>.

2.8.8 Path to Any-Ordered Set

If you place data to a structured event element of an any-ordered set by number (instead of by name), that number is related to the order of the members of the set as specified in the event definition, not to the order of the structured event elements as they occur if mapped with the event data. For example:

```
(define anyorder-struct (event-convert
(quote
(anyorder AS 1 1 und und 0 0
    (A ON 1 1 "abc" "abc" 0 0)
    (B ON 1 1 "def" "def" 0 0)
  )
)))
```

For this event definition, the path:

```
~input%anyorder.0
```

accesses the structured event element A, whether or not A occurs as the first element of the set “anyorder.”

2.9 Sample Programs

These sample programs give you a basic understanding of how to write Monk programs. Refer to the comments for an explanation of each program.

Example 1

```
;run this test case as follows:
;stctrans -ims Sample1.dat,Sample2.dat Sample.txt
;expected results
;Parsed data successfully
;Call procedure successfully
;
;to see full trace of the run issue the following command:
;stctrans -md -ims Sample1.dat,Sample2.dat Sample.txt


;define a simple funcition to get the length of data contained in the
;second input string
(define call-function
 (lambda ()
  (string-length input-string2) ;; 2nd input data file that gets
passed in
  (display "Call function successfully\n")
))


;delimiters used by our simple structure below
(define delimiter
 '( ("|")
    ("^")
))

;define simple structure root with 2 children child_0 and child_1
(define structure ($resolve-event-definition (quote
 (root ON 1 1 und und und und
  (child_0 ON 1 1 "one" und und und)
  (child_1 ON 1 1 "two" und und und)
 )
)))

;define input and output structures
(define input ($make-event-map delimiter structure))
```

```
(define output ($make-event-map delimiter structure))

;parser input data from string 1 and map to our simple structure
($event-parse input input-string1) ;; Input data file that gets
passed in

;should display parsed successfully if we used Sample1.dat
(display "Parsed data successfully\n")

;call the function defined above
(call-function)
```

Example 2

```
;Sample of Delimited Event Definition Structure
(define all_node_types-delm '(
 ("\r" endofrec)
 ("|" separator)
 ("~" array)
 ("^" separator)
 ("&" separator)
 ))

;Global Template Reference
(load "your.ssc")
(load "HL7/HL7_2.2/hl7_2.2_acc.ssc")
;End Global Template Reference

;Local Template Definition
(define Internal_Template ($resolve-event-definition (quote
  (Internal_Template ON 1 1 und und und -1
    (unnamed_1 ON 1 1 und und und -1)
    (unnamed_2 ON 1 1 und und und -1)
    (unnamed_3 ON 1 1 und und und -1)
  )
)))
;End Local Template Definition
```

Example 3

```
;MsgStructure Definition
(define all_node_types-struct ($resolve-event-definition (quote
  (all_node_types ON 1 1 und und und -1
    ((endofrec) fixed_examples ON 1 1 und und und -1
      (fixed_offset_length OF 1 1 und und 3 10)
      (fixed_pos OF 1 1 und und 3 ( 19 3))
      (fixed_any_order OF 1 1 und und 3 10)
    )
    (delimited_examples ON 1 1 und und und -1
      ((endofrec) non_repeating_delimited ON 1 1 und und und -1)
      ((endofrec) non_repeating_tagged ON 1 1 "InputTag"
"OutputDefaultData" und -1)
      ((endofrec) optional ON 0 1 und und und -1)
      ((endofrec) optional_repeating ON 0 INF und und und -1)
      ((endofrec) repeating ON 1 INF und und und -1)
      ((endofrec) range ON 5 10 und und und -1)
      (delimited_any_order AN 1 1 und und und -1)
    )
    ((endofrec) set_examples ON 1 1 und und und -1
      (ordered_set OS 1 1 und und und -1)
      (unordered_set AS 1 1 und und und -1)
      (ordered_separator_delim ONA 1 1 und und und -1)
      (ordered_repeating OS 1 INF und und und -1)
```

```
      (((Bd "BeginDelim") (Ed "EndDelim") endofrec required (Ri (Bd
"BeginRep") (Ed "EndRep")
required)) overriden_delims ON 1 1 und und und -1)
    )
    ((endofrec) template_example ON 1 1 und und und -1
      (your GTF 1 1 "your.ssc" your-struct und und)
      (Internal_Template LTN 1 1 und Internal_Template und und)
      (ACC GTN 1 1 HL7/HL7_2.2/hl7_2.2_acc.ssc" ACC-struct und und)
    )
  )
)))
;End MsgStructure Definition
```

Example 4

```
;Fixed MsgStructure Definition
(define fixed-struct ($resolve-event-definition (quote
  (fixed OF 1 1 und und und 0
    (fixed_len_offset OF 1 1 und und 3 3)
    (fixed_encoded_length OF 1 1 und und 5 ( 7 20))
    (unnamed_3 OF 1 1 und und und 0)
  )
)))
;End MsgStructure Definition
```

Chapter 3

Control Flow and Boolean Expressions

3.0.1 Overview

Control Flow Expressions control the order of statement execution. They include conditional, iteration and sequencing expressions.

Conditional expressions are used to test, compare, and selectively evaluate subordinate expressions. Conditional expressions are:

case on page 59

case-equal on page 60

cond on page 61

if on page 65

Iteration expressions evaluate subordinate expressions repeatedly according to specified conditions and include:

do on page 62

do* on page 64

The sequencing expression groups subordinate expressions for evaluation in a specified order. The sequencing expression is:

begin on page 58

Boolean expressions operate on zero or more arguments and return a Boolean value. They are often used in conjunction with conditional and iteration expressions to cause a particular branch of code to execute over alternates. The Boolean operators are:

and on page 57

or on page 67

not on page 66

and

Syntax

```
(and test1 test2 ...)
```

Description

and is a multi-conditional expression that evaluates left to right.

Parameters

Name	Type	Description
test*N*	expression	The expression to evaluate.

Return Value

The **and** expression stops processing and returns the result of the first test that returns false. If all expressions return true, not **#f**, the expression returns the result of the last expression evaluated. If no tests are listed, the **#t** is the result.

Examples

```
(define three-digit-string?               ; begin define
 (lambda (s)                              ; begin lambda on strings
  (and                                    ; begin and
   (string? s)                            ; test if s is string
   (= (string-length s) 3)                ; test if s has length of 3
   (char-numeric? (string-ref s 0))       ; test if 1st char is numeric
   (char-numeric? (string-ref s 1))       ; test if 2nd char is numeric
   (char-numeric? (string-ref s 2))       ; test if 3rd char is numeric
  )                                       ; end and
 )                                        ; end lambda
)                                         ; end define
```

begin

Syntax

```
(begin expression1 expression2 ...)
```

Description

Sequences evaluation of expressions. The expressions following **begin** are evaluated left to right.

Parameters

Name	Type	Description
expression	expression	The expression to evaluate.

Return Value

This expression returns the result of the evaluation of the last expression.

Examples

```
(define x 0)                            ; create variable x
(begin
  (set! x 5)                            ; change value of x
  (+ x 1)                               ; modify value of x
)
    => 6                                ; result


(begin
  (display "4 plus 1 equals ")          ; start display
  (display (+ 4 1))                     ; continue display
)

    => 4 plus 1 equals 5                ; result of display
```

case

Syntax

```
(case key
  ((datum11 datum12 ...) expression11 expression12 ...)
  ...
  ((datumn1 datumn2 ...) expressionn1 expressionn2 ...)
)
```

or

```
(case key
  ((datum11 datum12 ...) expression11 expression12 ...)
  ...
  (else expressionn1 expressionn2 ...)
)
```

where

key

can be any expression.

Description

Flow control expression. In operation, *key* is evaluated, and its result is compared against each datum in each clause using the **eqv?** procedure. There must be a minimum of one expression and one datum. If the result of the evaluation is found to be true (not **#f**), the expressions in that clause are evaluated left to right, and the result of the last expression is returned as the result of the **case** expression.

However, if the result is found to be different from every datum in the clause, there are two possible results:

1. If the last clause in the series is an **else** clause which has the form:

   (else $expression_{n1}$ $expression_{n2}$ ...)

   the expressions in the **else** clause are evaluated and the result of the last expression is returned as the result of the **case** expression.

2. If the last clause in the series is *not* an **else** clause, the result of the **case** expression is unspecified.

Parameters

None.

Return Value

Results of the evaluation of an expression associated with a particular datum.

Examples

```
(case (* 1 3)
    ((2 3 5 7) "prime")
    ((1 4 6 8 9) "composite")
)                                ==> "prime"
```

case-equal

Syntax

```
(case-equal key
  ((datum11  datum12 ...) expression11  expression12 ...)
  ...
  ((datumn1 datumn2 ...) expressionn1 expressionn2 ...)
)
```

or

```
(case-equal key
  ((datum11  datum12 ...) expression11  expression12 ...)
  ...
  (else expressionn1 expressionn2 ...)
)
```

where

key

can be any expression.

Description

Flow control expression. In operation, *key* is evaluated, and its result is compared against each datum in each clause using the **equal?** procedure. There must be a minimum of one expression and one datum. If the result of the evaluation is found to be true (not **#f**), the expressions in that clause are evaluated left to right, and the result of the last expression is returned as the result of the **case-equal** expression.

However, if the result is found to be different from every datum in the clause, there are two possible results:

1. If the last clause in the series is an **else** clause which has the form:

   (else *expression*$_{n1}$ *expression*$_{n2}$ ...)

   the expressions in the **else** clause are evaluated and the result of the last expression is returned as the result of the **case** expression.

2. If the last clause in the series is *not* an **else** clause, the result of the **case** expression is unspecified.

Parameters

None.

Return Value

Results of the evaluation of an expression associated with a particular datum.

Examples

```
(define var #\3)
(case-equal var
  ((#\1 #\3 #\5 #\7 #\9) "An ODD digit")
  ((#\0 #\2 #\4 #\6 #\8) "An EVEN digit")
  (else "Not a digit")
)       => "An ODD digit"
```

cond

Syntax

```
(cond
  ((test1) (expr11) (expr12) ...)
  ...
)
```

or

```
(cond
  ((test1) (expr11) (expr12) ...)
  ...
  (else (exprN1) (exprN2) ...)
)
```

Description

Flow control expression. The *test* expressions of the successive clauses are evaluated left to right until one of them evaluates to **#t** or to an expression equivalent to **#t**. After a *test* is found which evaluates to true, the remaining expressions of the clause are evaluated in order. The result of the last expression in the clause is returned as the result of the **cond** expression. For every test, there has to be at least one expression.

1. If the clause contains only a test but no expressions, the result of the test is returned as the result of the **cond** expression.
2. If the last clause in the series is an **else** clause, and no prior test evaluated to true, then the expressions in the **else** clause are evaluated and the result of the last expression is returned as the result of the **cond** expression.
3. If the last clause in the series is not an **else** clause, and no prior test evaluated to true, the result of the **cond** expression is unspecified.

Parameters

None.

Return Value

Returns unspecified if no conditions match. Else, returns the result of the valuation of the final expression in the test expression list.

Examples

```
(cond
  (( > 3 2) "greater")        ; evaluates to #t
  (( < 3 2) "less")           ; never evaluated
)                             ; end cond

(cond
  (( > 3 3) "greater")        ; evaluates to #f
  (( < 3 2) "less")           ; evaluates to #f
  (else "equal")              ; so the else is evaluated.
)                             ; end cond
```

do

Syntax

```
(do ((variable init increment) ...)
    (test result)
    body
)
```

Description

Executes a body of statements iteratively.

The **do** expression has three parts: the declaration of loop variables, the test expression and the body.

First, **do** creates zero or more variables, and binds them to the evaluation of their *init* expressions. Then, **do** executes the *test* expression.

If the result of the *test* expression is **#f**, *body* expressions are evaluated in order. Then the *increment* expressions are evaluated, the increment values are stored in the bound locations of the loop variables and *test* is evaluated again.

If the result of the *test* expression is **#t** or equivalent to **#t**, the *result* expression is evaluated and the do loop is complete.

Parameters

Name	Type	Description
triplet	list of two or three elements	The variable init increment statement. The increment portion is optional.
test	expression	The test to evaluate.
result	variable	The expressions to be evaluated if the test returns not **#f**. Optional.
body	expression	The expressions to be evaluated if the test returns **#f**.

Return Value

The value of the **do** expression is the value of the *result* expression if it exists. Otherwise the value is unspecified.

Example

```
(define str "MIXEDcase")
(do
  ( (i 0 (+ i 1)) )
  ((or (= i (string-length str))
       (char-lower-case? (string-ref str i))
  )
  i)
)       => 5
```

This code calculates the index of the first lower case character in the string *str*. In this case, the character “c” is the first lower case character and its index is 5. (Recall that strings are indexed starting from 0.)

The index variable is *i* which is initialized to zero and incremented by 1 at each step. The return value is also *i*. The body of this do loop is empty. All the work in this example is accomplished in the *test* and *result* portions of the do-loop.

do*

Syntax

```
(do* ((variable1 init1 increment1) ... )
     (test result)
     body
)
```

Description

Executes a body of statements iteratively.

The **do*** expression has three parts: the declaration of loop variables, the test expression and the body.

First, **do*** creates zero or more variables, and binds them to the evaluation of their *init* expressions. Then, **do*** executes the *test* expression.

If the result of the *test* expression is **#f**, *body* expressions are evaluated in order. Then the *increment* expressions are evaluated, the increment values are stored in the bound locations of the loop variables and *test* is evaluated again.

If the result of the *test* expression is **#t** or equivalent to **#t**, the *result* expression is evaluated and the do loop is complete.

do* operates just like the **do** expression with the exception that the bindings in **do*** are evaluated in order, and are available in subsequent bindings.

Parameters

Name	Type	Description
triplet	list of two or three elements	The variable init increment statement. The increment portion is optional.
test	expression	The test to evaluate.
result	variable	The expressions to be evaluated if the test returns not **#f**. Optional.
body	expression	The expressions to be evaluated if the test returns **#f**.

Return Value

The value of the **do*** expression is the value of the *result* expression if it exists. Otherwise the value is unspecified.

Example

```
(define ret "MIXEDcase")
(do ( (i 0 (+ i 1)) )
    ((or (= i (string-length ret))
        (char-lower-case? (string-ref ret i))
    ) i)
)              ==> 5
```

if

Syntax

```
(if test consequence alternative)
```

Description

Conditional construct used for flow control.

In the **if** expression, the *test* is evaluated. If the *test* returns anything other than **#f**, then the *consequence* is evaluated. If the *test* returns **#f**, then the *alternative* is evaluated.

Alternative is optional and may be omitted.

Parameters

Name	Type	Description
test	expression	The expression to be evaluated.
consequence	expression	The expression to be evaluated if the test returns true (not **#t**).
alternative	expression	The expression to be evaluated if the test returns **#f**. Optional.

Return Value

The result of the evaluation of the *consequence* or of the *alternative*. If the *test* returns a false value and no *alternative* is specified, then the result of the if expression is unspecified.

Example

```
(if                                ; begin if
  (> 3 2)                          ; test
  "test evaluates to #t"           ; consequence (then)
  "test evaluates to #f"           ; alternate
)       ==> "test evaluates to #t"
```

In this example, because 3 is greater than 2, the consequence, not the alternate is evaluated.

not

Syntax

```
(not obj)
```

Description

Determines if the object is false.

Parameters

Name	Type	Description
obj	any	The object to test for Boolean **#f**.

Return Value

Boolean

If the object is **#f**, the return value is **#t**. Else, the return is **#f.**

Examples

```
(not #t)          =>    #f
(not #f)          =>    #t
(not "a")         =>    #f
(not '(a b c))    =>    #f
```

or

Syntax

```
(or test1 test2 ....)
```

Description

Multi-conditional expression. **or** returns the result of the first test that evaluates to **#t** or to a value equivalent to **#t**.

Parameters

Name	Type	Description
test	expression	The expression to test.

Return Value

If all tests evaluate to **#f**, the expression returns **#f**. If no tests are done, returns **#f.**

Examples

```
(define empty-string          ; begin define
  (lambda (s)                 ; begin lambda on string s
    (and                      ; begin and
      (or                     ; begin or
        (string? s)           ; test if s is a string, else #f
        (path? s)             ; test if s is a path, else
                              ; returns #f
      )                       ; end or
      (zero? string-length s))              ; test if length is zero
    )                         ; end and
  )                           ; end lambda
)                             ; end define
```

Chapter 4

Definition, Binding and Assignment

Definition expressions create and manage global variables. They include:

Binding forms are expressions used to create local variables with local scopes and bind new values to the variables. They include:

Assignment expressions are used to assign new values into existing variables. They include:

define

Syntax

```
(define variable expression)
```

Description

Creates a new symbol equivalent to the evaluation of an expression.

The **define** function may be used to define procedures for later evaluation or to define symbols that evaluate to a given constant value.

You cannot use **define** to change the way Monk interprets keywords such as **do**, **case**, **if**, **filename**, and so forth.

To remove the declaration of a symbol, use the **undefine** function. See **"undefine" on page 76**.

Parameters

Name	Type	Description
variable	symbol	The symbol to be bound.
expression	expressions	The procedure being defined.
formals	symbols	The newly allocated list of actual arguments.
formal	single symbol	The list of all arguments.
body	expressions	The list of expressions to be evaluated.

Return Value

The return value is unspecified.

Examples

```
(define add3
  (lambda (x) (+ x 3))
)
(add3 5)        => 8
```

Add3 is created and is defined as the value of a **lambda** expression. A **lambda** expression returns a procedure so add3 is a procedure. Anytime after this **define** is executed, add3 can be invoked like any other function. When passed the value 3, the expression evaluates to 8. The capabilities of Monk are extended through this mechanism of defining functions.

```
(define y 7)
(add3 y)        => 10
```

In the second example, **y** is defined to have a constant value of 7. The symbol **y** can be passed to the function previously defined to generate the desired result.

defined?

Syntax

```
(defined? symbol)
```

Description

Determines if the symbol is defined globally or in the current environment.

Parameters

Name	Type	Description
symbol	symbol	The symbol to test for binding.

Return Value

This expression returns true **#t** if the symbol is bound; otherwise, it returns false **#f**.

Examples

```
(defined? x)         => #f

(defined  x 10)
(defined? x)         => #t
```

let

Syntax

```
(let bindings body)
```

where *bindings* have the form:

```
((variable1 init1) ...)
```

and *body* is a set of expressions.

Description

Creates bound variables of local scope.

The *init*s are evaluated in the current environment (in unspecified order), the *variable*s are bound to fresh locations holding the results, and the *body* is evaluated in the extended environment. The value of the last expression of the *body* is returned as the value of the **let** expression.

Parameters

Name	Type	Description
bindings	expression	Each *init* is an expression. It is an error for a *variable* to appear more than once in the list of variables being bound.
body	expression	Sequence of one or more expressions.

Return Value

The result of the evaluation the final expression in the *body*.

Examples

```
(let
  ((x 3) (y 7))
  (* x y)
)              => 21

(let
  ((x 2) (y 3)
    (let
      ((x 7)          ; variable x is bound to the value of 7
       (z (+ x y))    ; but for z, the old value of x (2) is used
      (* z x)         ; but here the new value of x (7) is used
    )
  )
)           => 35
```

Even though **x** is being bound to a location containing the number 7, this binding is not yet visible to the expression used for binding of variable **z**. Thus when **z** is bound, the expression is evaluated using the *old* value of the variable **x**, namely two.

let*

Syntax

```
(let* bindings body)
```

where *bindings* have the form:

```
((variable1 init1) ...)
```

Description

Creates bound variables of local scope.

The *init*s are evaluated in the current environment sequentially from left to right, the *variable*s are bound to fresh locations holding the results, and the *body* is evaluated in the extended environment. The value of the last expression of the *body* is returned as the value of the **let*** expression.

It operates just like the **let** expression with the exception that the *bindings* in **let*** are evaluated sequentially from left to right, and are available to subsequent *bindings*.

Parameters

Name	Type	Description
bindings	expression	Each *init* is an expression. It is an error for a *variable* to appear more than once in the list of variables being bound.
body	expression	A sequence of one ore more expressions.

Return Value

The result of the evaluation the final expression in the *body*.

Example

```
(let
  ((x 2) (y 3))
  (let*
    ((x 7)
     (z (+ x y)))        ; z sees the new value of x (7)
    (* z x)
  )
)            => 70
```

Because **let*** is used, the binding for **z** sees the new value of **x**, namely 7. The result is that **z** holds the value 10 and the final expression evaluates to 70.

letrec

Syntax

```
(letrec bindings body)
```

where *bindings* have the form:

```
((variable1 init1) ...)
```

Description

Creates bound variables of local scope. The variables are not evaluated until they are actually called.

The *init*s are evaluated in the current environment sequentially from left to right, the *variable*s are bound to fresh locations holding the results, and the *body* is evaluated in the extended environment. The value of the last expression of the *body* is returned as the value of the **letrec** expression.

It operates just like the **let*** expression where the *bindings* in **let*** are evaluated sequentially from left to right, and are available to subsequent *bindings*.

Parameters

Name	Type	Description
bindings	expression	Each *init* is an expression. It is an error for a *variable* to appear more than once in the list of variables being bound.
body	expression	A sequence of one ore more expressions.

Return Value

The result of the evaluation the final expression in the *body*.

Example

```
(letrec ((x 2)(y(+ x 3)))
   (display y)
)
    =>    5
```

set

Syntax

```
(set symbol_var expression)
```

Description

Evaluates the *symbol_var* parameter and the *expression* parameter, and then binds the resulting expression to the resulting symbol.

Name	Type	Description
symbol_var	symbol	The variable to set as the result of the evaluation of the expression.
expression	expression	One or more expressions to evaluate.

Return Value

Returns the result of an evaluated expression.

Example

```
(define hello "")
(define abc (string->symbol "hello"))
(set abc "goodbye")
(display abc)                  => hello
(display hello)                => goodbye
```

set!

Syntax

```
(set! variable expression)
```

Description

Evaluates the *expression* parameter and binds the result to the *variable*.

Parameters

Name	Type	Description
variable	symbol	The variables to set as the result of the evaluation of the expression.
expression	expression	One or more expressions to evaluate.

Return Value

Returns the result of an evaluating expression.

Example

```
(define x 0)                    ; create variables
(set! x "Hello")                ; change value of x

=> "Hello"                      ; result
```

undefine

Syntax

```
(undefine symbol)
```

Description

Removes the declaration of a symbol. For information on declaring symbols, see **"define" on page 69**.

Parameters

Name	Type	Description
symbol	symbol	The name of the symbol to be undefined.

Return Value

If successful, returns the value of the symbol that was undefined. Otherwise, returns {MONK_UNBOUND}.

Example

```
(define new_symbol "Hello")       ;defines the symbol
(display new_symbol)              => Hello
(newline)
(display (defined? new_symbol))   => =>t
(newline)
(undefine new_symbol)             ;undefines the symbol
(display (defined? new_symbol))   => =>f
```

Character Functions

A character is a fundamental data type containing the representation of a single character within the machine's character set.

A character is identified by preceding it with **#\.** To indicate any single printable character, precede it by #\. For example, #\a, #\b, #\c, ... #\A, #\B, #\C, ... \#1, \#2, ... To identify special characters the preferred method is to use the name of the character, for example #\space, #\tab.

Character functions which performs conversion to or from other data types may be found in **Conversion Procedures** on page 244.

Following is a list of functions which operate on a character:

char?

Syntax

```
(char? parm)
```

Description

Tests the supplied parameter to determine whether or not it is a character.

Parameters

Name	Type	Description
parm	any	The object to check.

Return Value

Boolean

Returns a **#t** if the parameter is a valid character. Otherwise, returns **#f**.

Examples

```
(char? #\k)                     => #t

(char? "z")                     => #f
```

"z" is not a character. It is a string because it is contained within double quotes.

```
(char? 137)                     => #f

(char? #\1)                     => #t

(char? #\formfeed)              => #t

(char? (string-ref "a b c" 2))  => #t
```

char=?

Syntax

```
(char=? char1 char2)
```

Description

Compares two characters for equality. This function is case sensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean
Returns **#t** if *char1* is the same as *char2*. Otherwise, returns **#f**.

Examples

```
(char=? #\3 #\3)     => #t

(char=? #\3 #\4)     => #f

(char=? #\a #\A)     => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char<?

Syntax

```
(char<? char1 char2)
```

Description

Compares two characters for order. This function is case sensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is less than *char2* within the character collation sequence. Otherwise, returns **#f**.

Examples

```
(char<? #\3 #\3)              => #f

(char<? #\3 #\4)              => #t

(char<? #\a #\A)              => #f

(char<? #\a #\b)              => #t
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char>?

Syntax

```
(char>? char1 char2)
```

Description

compares two characters for order within the character collation sequence. This function is case sensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is greater than *char2*. Otherwise, returns **#f**.

Examples

```
(char>? #\3 #\3)            => #f

(char>? #\4 #\3)            => #t

(char>? #\a #\A)            => #t

(char>? #\a #\a)            => #f

(char>? #\a #\b)            => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char<=?

Syntax

```
(char<=? char1 char2)
```

Description

Compares two characters for order within the character collation sequence or for equality. This function is case sensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is less than or the same as *char2*. Otherwise, returns **#f**.

Examples

```
(char<=? #\3 #\3)            => #t

(char<=? #\3 #\4)            => #t

(char<=? #\a #\A)            => #f

(char<=? #\a #\a)            => #t

(char<=? #\a #\b)            => #t
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char>=?

Syntax

```
(char>=? char1 char2)
```

Description

Compares two characters for order within the character collation sequence or for equality. This function is case sensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is greater than or the same as *char2*. Otherwise, returns **#f**.

Examples

```
(char>=? #\3 #\3)    => #t

(char>=? #\3 #\4)    => #f

(char>=? #\a #\A)    => #t

(char>=? #\a #\a)    => #t

(char>=? #\a #\b)    => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-ci=?

Syntax

```
(char-ci=? char1 char2)
```

Description

Determines if the two specified characters are equal. This function is not case sensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is the same as *char2*. Otherwise, returns **#f**.

Examples

```
(char-ci=? #\3 #\3)     => #t

(char-ci=? #\3 #\4)     => #f

(char-ci=? #\a #\A)     => #t

(char-ci=? #\a #\a)     => #t
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-ci<?

Syntax

```
(char-ci<? char1 char2)
```

Description

Compares two characters for order. This function is case insensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is the same as *char2*. Otherwise, returns **#f**.

Examples

```
(char-ci<? #\3 #\3)     => #f

(char-ci<? #\3 #\4)     => #t

(char-ci<? #\a #\A)     => #f

(char-ci<? #\a #\a)     => #f

(char-ci<? #\a #\b)     => #t
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-ci>?

Syntax

```
(char-ci>? char1 char2 )
```

Description

Compares two characters for order. This function is case insensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is the greater than *char2*. Otherwise, returns **#f**.

Examples

```
(char-ci>? #\3 #\3)      => #f

(char-ci>? #\4 #\3)      => #t

(char-ci>? #\a #\A)      => #f

(char-ci>? #\a #\a)      => #f

(char-ci>? #\a #\b)      => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-ci<=?

Syntax

```
(char-ci<=? char1 char2)
```

Description

Compares two characters for being less or equal. This function is case insensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is less or the same as *char2*. Otherwise, returns **#f**.

Examples

```
(char-ci<=? #\3 #\3)      => #t

(char-ci<=? #\4 #\3)      => #f

(char-ci<=? #\a #\A)      => #t

(char-ci<=? #\a #\a)      => #t

(char-ci<=? #\a #\b)      => #t
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-ci>=?

Syntax

```
(char-ci>=? char1 char2)
```

Description

char-ci>? compares two characters for order. This function is not case sensitive.

Parameters

Name	Type	Description
char1	character	Initial character for the comparison.
char2	character	Second character for comparison.

Return Value

Boolean

Returns **#t** if *char1* is the same as *char2*. Otherwise, returns **#f**.

Examples

```
(char-ci>=? #\3 #\3)     => #t

(char-ci>=? #\3 #\4)     => #f

(char-ci>=? #\a #\A)     => #t

(char-ci>=? #\a #\a)     => #t

(char-ci>=? #\a #\b)     => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-alphabetic?

Syntax

```
(char-alphabetic? char)
```

Description

Determines whether or not the specified character is an alphabetic character.

Parameters

Name	Type	Description
char	character	The character to compare.

Return Value

Boolean

Returns **#t** if the specified character is alphabetic. Otherwise, returns **#f**.

Examples

```
(char-alphabetic? #\a)          => #t
(char-alphabetic? #\;)          => #f
(char-alphabetic? #\3)          => #f
```

char-and

Syntax

```
(char-and char1 char2)
```

Description

Returns a new character which is the Boolean **and** operation on the specified character.

Parameters

Name	Type	Description
char1	character	Initial character for the **and** operation.
char2	character	Second character for the **and** operation.

Return Value

character
Returns a character representing the result of the Boolean **and** on the specified characters.

Example

```
(char-and #\G #\C)        => C
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-downcase

Syntax

```
(char-downcase char)
```

Description

Converts the specified character from upper case to lower case.

Parameters

Name	Type	Description
char	character	The character to convert.

Return Value

char

Returns a lower case character for any alphabetic character found.

Examples

```
(char-downcase #\A)      => #\a

(char-downcase #\a)      => #\a

(char-downcase #\3)      => #\3
```

char-lower-case?

Syntax

```
(char-lower-case? char)
```

Description

Tests the specified character to determine whether or not it is a lowercase alphabetic character.

Parameters

Name	Type	Description
char	character	The character to test.

Return Value

Boolean

Returns **#t** if the specified character is a lowercase alphabetic character. Otherwise, returns **#f**.

Examples

```
(char-lowercase? #\A)      => #f

(char-lowercase? #\a)      => #t

(char-lowercase? #\3)      => #f

(char-lowercase? #\;)      => #f

(char-lowercase? #\))      => #f
```

char-not

Syntax

```
(char-not char)
```

Description

Returns a new character which is the Boolean **not** operation on the specified character.

Parameters

Name	Type	Description
char	character	Character for performing the **not** operation.

Return Value

character
Returns a character representing the result of the Boolean **not** operation on the specified character.

Example

```
(char-not #\G)  => #\, (comma)
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-numeric?

Syntax

```
(char-numeric? char)
```

Description

Determines whether the specified character is numeric.

Parameters

Name	Type	Description
char	character	The character to test.

Return Value

Boolean

Returns **#t** if the specified character is numeric. Otherwise, returns **#f**.

Examples

```
(char-numeric? #\A)     => #f

(char-numeric? #\a)     => #f

(char-numeric? #\3)     => #t

(char-numeric? #\;)     => #f

(char-numeric? #\))     => #f
```

char-or

Syntax

```
(char-or char1 char2)
```

Description

Returns a new character which is the Boolean **or** on the two specified characters.

Parameters

Name	Type	Description
char1	character	Initial character for the **or** operation.
char2	character	Second character for the **or** operation.

Return Value

character
Returns a character representing the result of the Boolean **or** on the specified characters.

Examples

```
(char-or #\G #\C)      => #\G
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-shift-left

Syntax

```
(char-shift-left char num)
```

Description

Returns a new character which the left shift of the bits representing the specified character and performs the shift operation the number of times specified by the second parameter.

Parameters

Name	Type	Description
char	character	Initial character for the shift operation.
num	integer	Number of times to perform the shift left.

Return Value

character
Returns a character representing the result of the shift operation on the specified character.

Examples

```
(char-shift-left #\G 3)     => #\9
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-shift-right

Syntax

```
(char-shift-right char num)
```

Description

Returns a new character which is the right shift of the bits representing the specified character and performs the shift operation the number of times specified by the second parameter.

Parameters

Name	Type	Description
char	character	Initial character for the shift operation.
num	integer	Number of times to perform the shift right.

Return Value

character
Returns a character representing the result of the shift operation on the specified character.

Examples

```
(char-shift-right #\G 3)      => #\x
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-type

Syntax

```
(char-type char)
```

Description

Determines the type of the specified character.

Parameters

Name	Type	Description
char	character	A character.

Return Value

symbol

Returns one of the following encoding types:

:1bBIG5	:2Byte
:1bEUC	:3bUTF8
:1bSJIS	:3Byte
:1bUHC	:4bUTF16
:1bUTF8	:4bUTF8
:1Byte	:4Byte
:2bBIG5	:ASCII
:2bEUC	:BIG5
:2bSJIS	:EBCDIC
:2bUHC	:UCS2
:2bUTF16	:UTF16
:2bUTF8	

Examples

```
(define mychar (integer->char 100))
(char-type mychar)                =>  :ASCII
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

char-type!

Syntax

```
(char-type! type char)
```

Description

Sets the character type for a character.

Parameters

Name	Type	Description
type	symbol	One of the following character types: ▪ :1bBIG5 ▪ :1bEUC ▪ :1bSJIS ▪ :1bUHC ▪ :1bUTF8 ▪ :1Byte ▪ :2bBIG5 ▪ :2bEUC ▪ :2bSJIS ▪ :2bUHC ▪ :2bUTF16 ▪ :2bUTF8 ▪ :2Byte ▪ :3bUTF8 ▪ :3Byte ▪ :4bUTF16 ▪ :4bUTF8 ▪ :4Byte ▪ :ASCII ▪ :BIG5 ▪ :EBCDIC ▪ :UCS2 ▪ :UTF16
char	character	The character whose type you want to set.

Return Value

character
Returns the character whose type has been set.

Examples

```
(char-type (char-type! :EBCDIC #\a))      =>    :EBCDIC

(char-type! :EBCDIC #\a)                  =>    a

(char-type! :2Byte #\a)                   =>     a

(char-type! :4Byte #\a)                   =>       a

(char-type! :DogByte #\a)                 =>    {MONK_EXCEPTION}

(char? (char-type! :2Byte #\a))            =>    #t
```

char-type?

Syntax

```
(char-type? type char)
```

Description

Determines whether specified character is of the specified type.

Parameters

Name	Type	Description
type	symbol	One of the following: ▪ :1bBIG5 ▪ :1bEUC ▪ :1bSJIS ▪ :1bUHC ▪ :1bUTF8 ▪ :1Byte ▪ :2bBIG5 ▪ :2bEUC ▪ :2bSJIS ▪ :2bUHC ▪ :2bUTF16 ▪ :2bUTF8 ▪ :2Byte ▪ :3bUTF8 ▪ :3Byte ▪ :4bUTF16 ▪ :4bUTF8 ▪ :4Byte ▪ :ASCII ▪ :BIG5 ▪ :EBCDIC ▪ :UCS2 ▪ :UTF16
char	character	A character.

Return Value

Boolean

Returns **#t** if the character is of the specified type. Otherwise, it returns **#f**.

Examples

```
(define mychar (integer->char 100))

(char-type? :ASCII mychar)     => #t

(char-type? :EBCDIC mychar)    => #f
```

char-upcase

Syntax

```
(char-upcase char)
```

Description

Converts a character from lowercase to uppercase.

Parameters

Name	Type	Description
char	character	A character to convert.

Return Value

char
Returns an uppercase character for any alphabetic character found.

Examples

```
(char-upcase #\a)      => #\A

(char-upcase #\A)      => #\A

(char-upcase #\3)      => #\3

(char-upcase #\#)      => #\#
```

char-upper-case?

Syntax

```
(char-upper-case? char)
```

Description

Determines whether the specified character is an uppercase alphabetic character.

Parameters

Name	Type	Description
char	character	The character to test.

Return Value

Boolean
Returns **#t** if the character is an uppercase alphabetic character. Otherwise, returns **#f**.

Examples

```
(char-upper-case? #\A)       => #t

(char-upper-case? #\a)       => #f

(char-upper-case? #\3)       => #f

(char-upper-case? #\;)       => #f

(char-upper-case? #\)        => #f
```

char-whitespace?

Syntax

```
(char-whitespace? char)
```

Description

Determines whether the character is a blank space character.

Parameters

Name	Type	Description
char	character	The character to test.

Return Value

Boolean

Returns **#t** if the specified character is a blank character. Otherwise, returns **#f**.

Examples

```
(char-whitespace? #\ )       => #t

(char-whitespace? #\A)       => #f

(char-whitespace? #\b)       => #f

(char-whitespace? #\3)       => #f

(char-whitespace? #\;)       => #f
```

char-xor

Syntax

```
(char-xor char1 char2)
```

Description

Returns a new character which is the Boolean XOR (exclusive OR) on two specified characters.

Parameters

Name	Type	Description
char1	character	Initial character for the XOR operation.
char2	character	Second character for the XOR operation.

Return Value

character
Returns a character representing the result of the Boolean XOR on the specified characters.

Examples

```
(char-xor #\G #\C)      => \#space

(char-xor #\g #\C)      => \#$

(char-xor #\G #\c)      => \#$
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

Chapter 6

String Functions

A String is defined as a sequence of characters. Strings are denoted by characters within a pair of double quotation marks (" "). For example: “spot is a dog”, “1234” and “a # c” are all strings.

Strings cannot be modified if constants. Such strings are said to be immutable. For example, the following will fail because FirstName is immutable:

```
(define FirstName "Benny")
(string-set! FirstName 0 #\P)
```

To create a mutable string, use the **make-string** function. The code above will succeed if rewritten like this:

```
(define FirstName (make-string 1 "Benny"))
(string-set! FirstName 0 #\P)
```

The Monk functions operating on strings are listed on the next two pages in the table below:

format

Syntax

```
(format formatinstruction value)
```

Description

Converts *value* according to *formatinstruction*.

May be used to convert string data representing numbers to a variety of binary, octal, decimal or hexadecimal representations. Also used to convert Monk time objects and other Monk objects.

For a comprehensive list of examples, see **"Format Specification" on page 34**

Parameters

Name	Type	Description
format-spec	expression	The specification of the output format. The syntax for the format instruction is documented in **Format Specification** on page 34.
arg	string/path	A string (or path).

Return Value

The format expression takes a string and formats according to format-spec instruction and returns the formatted string as its result.

Examples

```
Input

(define str "string")
(format "%s-->end" str)          => "string-->end"
(format "%10s-->end" str)        => "    string-->end"

(define num "123456")
(format "%d-->end" num)          => "123456-->end"
(format "%10d-->end" num)        => "    123456-->end"

(define float "123.456")
(format "%f-->end" float)        => "123.456000-->end"
(format "%15f-->end" float)      => "     123.456000-->end"
```

htonl->string

Syntax

```
(htonl->string num)
```

Description

Converts a long integer from the host byte order to a string in network byte order.

Parameters

Name	Type	Description
num	integer	A long integer.

Return Value

string
Returns a string in two-byte network byte order.

Examples

```
(htonl->string 98)     => "   b"

(htonl->string 43)     => "   +"

(htonl->string 35)     => "   #"
```

htons->string

Syntax

```
(htons->string num)
```

Description

Converts a short (hex) integer from the host byte order to a string in network byte order.

Parameters

Name	Type	Description
num	integer	A short integer.

Return Value

string
Returns a string in two-byte network byte order.

Examples

```
(htons->string 98)    => " b"

(htons->string 43)    => " +"

(htons->string 35)    => " #"
```

list->string

Syntax

```
(list->string list)
```

Description

Concatenates a series of characters into a string.

Parameters

Name	Type	Description
list	list	A list of characters to concatenate into a string.

Return Value

string

Returns a string of the characters in the list.

Examples

```
(list->string '(#\a #\b #\c))                 => "abc"

(list->string '(#\T #\h #\i #\s))             => "This"

(list->string '(#\S #\T #\C #\  #\3 #\#))    => "STC 3#"
```

Note that '#\ ', which is the escape sequence for a space must be followed by *another* space in order to delimit the space character from the following character, #\3. Better style is to write this as

```
(list->string '(#\S #\T #\C #\space #\3 #\#))    => "STC 3#"
```

make-string

Syntax

```
(make-string nreps [fill-char|fill-str])
```

Description

Creates a new mutable string.

You may specify either a character or a string indicated. In either case, the new string is created with that character or string repeated *nreps* times.

If no *fillchar* is indicated, **make-string** defaults to creating *nreps* single-character bytes.

Typical usage for **make-string** is in conjunction with **define** resulting in the creation of a mutable string.

Parameters

Name	Type	Description
n-repetitions	integer	The number of repetitions of the fill character or string.
fill-char	character	The character that comprises the new string. Optional.
fill-str	character	The string that comprises the new string. Optional.

Return Value

string
Returns a string of characters.

Examples

```
(make-string 5 #\a)            => "aaaaa"

(make-string 4 #\4)            => "4444"

(make-string 2 "Hello! ")      => "Hello! Hello! "

(define name (make-string 1 "John"))
```

The variable **name** becomes a mutable string as a result of defining to be the result of **make-string**. It may be manipulated later with commands that change string length, pad the string, set characters or otherwise alter the contents of **name**.

regex

Syntax

```
(regex reg_exp string)
```

Description

Matches a string against a regular expression and returns **#t** if there is a match. Otherwise, returns **#f**.

Parameters

Name	Type	Description
reg_exp	expression	The regular expression to test.
string	string	The string to test against the regular expression.

Return Value

Boolean

Returns **#t** if the string does match the regular expression. Otherwise, returns **#f**.

Example

```
;compare Event Type Code to regular expression "A01"
(regex "A01" ~input%X12.EVN.ETC)

;compare message location to message location
(regex ~input%X12.PID.Policy_N ~input%X12.IN2.Insured_SSN )

;compare message location to message location where
;both locations are in repeating segments
(do
    ((i  0 (+  i  1)))
    ((>= i (count  ~input%X12.ORCGRP)))
  (do
      ((j  0 (+  j  1)))
      ((>= j (count ~input%X12.ORCGRP)))
    (if (regex  ~input%X12.ORCGRP[<i>].ORC.11
                ~input%X12.ORCGRP[<j>].RXR.2)
        (copy  ~input%X12.ORCGRP[<i>].OBXGRP.OBX.2
               ~output%MSG.DTM.<i>.0 "" )
        )
    )
  )
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string

Syntax

```
(string char...[char])
```

Description

Concatenates a series of individual characters into a string.

Parameters

Name	Type	Description
char	character	A series of characters. Minimum of one character.

Return Value

string
Returns a string consisting of the concatenated characters.

Examples

```
(string #\a #\b #\c)                    => "abc"

(string #\T #\h #\i #\s)                => "This"

(string #\S #\T #\C #\space #\3 #\#)   => "STC 3#"
```

string?

Syntax

```
(string? object)
```

Description

Determines whether the object is a string.

Parameters

Name	Type	Description
obj	any	The object to be tested.

Return Value

Boolean

Returns **#t** if the object is a string. Otherwise, returns **#f**.

Examples

```
(string? "This is a string")     => #t
(string? 17)                     => #f
(string? #\a)                    => #f
```

string<?

Syntax

```
(string<? string1 string2)
```

Description

Compares *string1* and *string2* for lexical order.

Lexical order is determined by comparing corresponding characters of both strings until a non-match occurs. If the non-matching character of *string1* is less than the non-matching character of *string2*, (in the sense of the **char<?** function) **#t** is returned. If greater, then **#f** is returned. Otherwise, **#f** is returned.

string<? is case sensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean

Returns **#f** if *string1* is less than *string2*. Otherwise, it evaluates to **#t**.

Examples

```
(string<? "SMITH" "SMITH")      => #f

(string<? "SMITH" "SMYTHE")     => #t

(string<? "SMITH" "SMITHY")     => #t

(string<? "2222" "2222")        => #f

(string<? "2222" "231")         => #t
```

Note that the comparison against “231” evaluates to **#t** because this is a lexical ordering. If this ordering were numeric, the previous example would evaluate to **#f**.

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string<=?

Syntax

```
(string<=? string1 string2)
```

Description

Compares *string1* and *string2* for lexical order.

Lexical order is determined by comparing corresponding characters of both strings until a non-match occurs. If the non-matching character of *string1* is greater than the non-matching character of *string2*, (in the sense of the **char>?** function) **#f** is returned. If less, then **#t** is returned. Otherwise, **#t** is returned.

string<=? is case sensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean

Returns **#t** if *string1* is less or equal to *string2*. Otherwise, it evaluates to **#f**.

Examples

```
(string<=? "SMITH" "SMITH")      => #t

(string<=? "SMITH" "SMYTHE")     => #t

(string<=? "SMITH" "SMITHY")     => #t

(string<=? "2222" "2222")        => #t

(string<=? "2222" "231")         => #t
```

Note that the comparison against “231” evaluates to **#t** because this is a lexical ordering. If this ordering were numeric, the previous example would evaluate to **#f**.

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string=?

Syntax

```
(string=? string1 string2)
```

Description

Compares *string1* and *string2* for equality. This function is case sensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean

Returns **#f** if any character in *string1* differs from its corresponding character in *string2*. Otherwise, it evaluates to **#t**.

Examples

```
(string=? "1234" "1234")    => #t

(string=? "1234" "1235")    => #f

(string=? "abcd" "abcd")    => #t

(string=? "abcd" "abCd")    => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string>?

Syntax

```
(string>? string1 string2)
```

Description

Compares *string1* and *string2* for lexical order.

Lexical order is determined by comparing corresponding characters of both strings until a non-match occurs. If the non-matching character of *string1* is less than the non-matching character of *string2*, (in the sense of the **char<?** function) **#f** is returned. If greater, then **#t** is returned. Otherwise, **#f** is returned.

string>? is case sensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean

Returns **#t** if *string1* is greater than *string2*. Otherwise, it evaluates to **#f**.

Examples

```
(string>? "1234" "1234")     => #f

(string>? "1234" "1233")     => #t

(string>? "abcd" "abcd")     => #f

(string>? "abcd" "abCd")     => #t

(string>? "2222" "2222")     => #f

(string>? "2222" "231")      => #f
```

Note that the comparison against “231” evaluates to **#f** because this is a lexical ordering. If this ordering were numeric, the previous example would evaluate to **#t**.

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string>=?

Syntax

```
(string>=? string1 string2)
```

Description

Compares *string1* and *string2* for lexical order.

Lexical order is determined by comparing corresponding characters of both strings until a non-match occurs. If the non-matching character of *string1* is less than the non-matching character of *string2*, (in the sense of the **char<?** function) **#f** is returned. Otherwise, **#t** is returned.

string>=? is case sensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean

Returns **#t** if *string1* is greater than or equal *string2*. Otherwise, it evaluates to **#f**.

Examples

```
(string>=? "1234" "1234")    => #t

(string>=? "1234" "1233")    => #t

(string>=? "abcd" "abcd")    => #t

(string>=? "abcd" "abCd")    => #t

(string>=? "2222" "2222")    => #t

(string>=? "2222" "231")     => #f
```

Note that the comparison against “231” evaluates to **#f** because this is a lexical ordering. If this ordering were numeric, the previous example would evaluate to **#t**.

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string-append

Syntax

```
(string-append string...stringN)
```

Description

Appends a list of specified strings to form a new string.

Parameters

Name	Type	Description
string...stringN	string	A series of strings to concatenate.

Return Value

string
Returns a new string consisting of the concatenated specified strings.

Example

```
(string-append "345" "012")                 => "345012"
```

string-append!

Syntax

```
(string-append! old_string new_ring)
```

Description

Appends a string of characters to an existing string and dynamically increases the size of the existing string to fit the new length.

Parameters

Name	Type	Description
old_string	string	The previously existing string symbol.
new_string	string	The string to append to the original string.

Return Value

string
Returns the newly created string with the additional string appended.

Examples

```
(define old_string (make-string 5 #\a))
(string-append! old_string " append this string")
(display old_string)
     =>      aaaaa append this string
```

string-checksum

Syntax

```
(string-checksum string)
```

Description

Calculates a successive XOR (exclusive OR) operation on all bytes in the specified string.

Parameters

Name	Type	Description
string	string	The string on which to perform the checksum.

Return Value

integer
Returns an integer representing the checksum of the string.

Examples

```
(string-checksum "ABCDEFGHIJKK")     => 11

(string-checksum "123")              => 48
```

string-ci=?

Syntax

```
(string-ci=? string1 string2)
```

Description

Compares *string1* and *string2* for equality without regard for case.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean

Returns **#f** if each character in *string1* is not the same as the corresponding character in *string2*. Otherwise, it evaluates to **#t**.

Examples

```
(string-ci=? "1234" "1234")                  => #t
(string-ci=? "1234" "1235")                  => #f
(string-ci=? "abcd" "abcd")                  => #t
(string-ci=? "abcd" "abCd")                  => #t
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string-ci<?

Syntax

```
(string-ci<? string1 string2)
```

Description

Compares *string1* and *string2* for lexical order without regard for case.

Lexical order is determined by comparing corresponding characters of both strings until a non-match occurs (in the sense of **char-ci=?** function). If the non-matching character of *string1* is less than the non-matching character of *string2*, (in the sense of the **char-ci<?** function) **#t** is returned. Otherwise, **#f** is returned.

string-ci<? is case insensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean
Returns **#t** if *string1* is less than *string2* without regard for case. Otherwise, it evaluates to **#f**.

Examples

```
(string-ci<? "1234" "1234")    => #f

(string-ci<? "1234" "1235")    => #t

(string-ci<? "abcd" "ABCD")    => #f

(string-ci<? "abcd" "ABCE")    => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string-ci>?

Syntax

```
(string-ci>? string1 string2)
```

Description

Compares *string1* and *string2* for lexical order without regard for case.

Lexical order is determined by comparing corresponding characters of both strings until a non-match occurs (in the sense of **char-ci=?** function). If the non-matching character of *string1* is less than the non-matching character of *string2*, (in the sense of the **char-ci<?** function) **#f** is returned. Otherwise, **#t** is returned.

string-ci>? is case insensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean
Returns **#t** if *string1* is greater than *string2* without regard for case. Otherwise, it evaluates to **#f**.

Examples

```
(string-ci>? "1234" "1234")    => #f

(string-ci>? "1234" "1233")    => #t

(string-ci>? "abcd" "ABCD")    => #f

(string-ci>? "abcd" "ABCC")    => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string-ci<=?

Syntax

```
(string-ci<=? string1 string2)
```

Description

Compares *string1* and *string2* for lexical order without regard for case.

Lexical order is determined by comparing corresponding characters of both strings until a non-match occurs (in the sense of **char-ci=?** function). If the non-matching character of *string1* is greater than the non-matching character of *string2*, (in the sense of the **char-ci>?** function) **#f** is returned. Otherwise, **#t** is returned.

string-ci<=? is case insensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean

Returns **#t** if *string1* is less than or equal to *string2* without regard for case. Otherwise, it evaluates to **#f**.

Examples

```
(string-ci<=? "1234" "1234")    => #t

(string-ci<=? "1234" "1233")    => #f

(string-ci<=? "abcd" "ABCD")    => #t

(string-ci<=? "abcd" "ABCC")    => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string-ci>=?

Syntax

```
(string-ci>=? string1 string2)
```

Description

Compares *string1* and *string2* for lexical order without regard for case.

Lexical order is determined by comparing corresponding characters of both strings until a non-match occurs (in the sense of **char-ci=?** function). If the non-matching character of *string1* is less than the non-matching character of *string2*, (in the sense of the **char-ci<?** function) **#f** is returned. Otherwise, **#t** is returned.

string-ci>=? is case insensitive.

Parameters

Name	Type	Description
string1	string	First string to test.
string2	string	Second string to test.

Return Value

Boolean

Returns **#f** if each character in *string1* is not greater than or the same as the corresponding character in *string2*. Otherwise, it evaluates to **#t**.

Examples

```
(string-ci<=? "1234" "1234")    => #t

(string-ci<=? "1234" "1233")    => #f

(string-ci<=? "abcd" "ABCD")    => #t

(string-ci<=? "abcd" "ABCC")    => #f
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

string-copy

Syntax

```
(string-copy source)
```

Description

Copies the source string.

Parameters

Name	Type	Description
source	string	The string to copy.

Return Value

string
Returns a copy of the specified source.

Examples

```
(string-copy "This is input")    => "This is input"

(define x "abc")
(set! x (string-copy "12345"))
(display x)
```

prints “12345” to the display

string-copy!

Syntax

```
(string-copy! dest-str char-pos copy-str)
```

Description

Modifies the destination string at the character position with the copy string.

The byte-length of the destination string and the copy string must be identical. The string length is self-expanding only when the byte length of the copy string exceeds that of the destination string at the end of a string. See the second example.

Parameters

Name	Type	Description
dest-str	string	The original string to be modified.
char-pos	integer	The character position where the modification begins.
copy-str	string	The new string to copy into the original string at the character position.

Return Value

string
Returns the modified string.

Examples

```
(define sentence (make-string "The house is blue"))
(string-copy! sentence 0 "Our")      => "Our house is blue"

(define sentence (make-string "The house is blue"))
(string-copy! sentence 13 "violet") => "The house is violet"
```

string-crc16

Syntax

```
(string-crc16 string)
```

Description

Calculates a cyclical redundancy check on all bytes in a string using the CRC-16 algorithm.

Parameters

Name	Type	Description
string	string or path	The string to check.

Return Value

integer
Returns the CRC of the specified string.

Examples

```
(string-crc16 "AAAAA")      => 61332

(string-crc16 "12345")      => 21612
```

string-crc32

Syntax

```
(string-crc32 string)
```

Description

Calculates a cyclical redundancy check on all bytes in a string using the CRC-32 algorithm.

Parameters

Name	Type	Description
string	string or path	The string to check.

Return Value

integer
Returns the CRC of the specified string.

Examples

```
(string-crc32 "AAAAA")   => 435704073

(string-crc32 "12345")   => -873121252
```

string-downcase

Syntax

```
(string-downcase source)
```

Description

Returns a copy of the source with all alphabetic characters converted to lower case.

Parameters

Name	Type	Description
source	string or path	The string to manipulate.

Return Value

string

Returns a copy of the source with all alphabetic characters converted to lower case.

Examples

```
(string-downcase "A String")       => "a string"

(string-downcase "AAA")            => "aaa"
```

string-empty?

Syntax

```
(string-empty? string)
```

Description

Checks to see whether a string is empty or not.

Parameters

Name	Type	Description
string	string	The string to be checked.

Return Value

string
Returns **#t** if the string is empty. Otherwise, returns **#f**.

Examples

```
(string-empty? "")    =>   #t

(string-empty? "x")   =>   #f
```

string-fill!

Syntax

```
(string-fill! string char)
```

Description

Replaces every character in the specified string with the specified character.

string must be mutable.

Parameters

Name	Type	Description
string	string	The string to manipulate. Must be a mutable string.
char	character	Character with which to fill the string.

Return Value

Unspecified.

Example

```
(define mystring (make-string 5))
(string-fill! mystring #\d)          => "ddddd"
```

The function **make-string** when combined with **define** will create a mutable string. Mutable strings can be have their contents changed.

string-insert!

Syntax

```
(string-insert! dest-str char-pos insert-str)
```

Description

Inserts a new string into an existing string.

The characters in the existing string are shifted right. *dest-str* must be mutable. This function does not alter the data on the original string.

Parameters

Name	Type	Description
dest-str	string	The original string to be modified.
char pos	integer	The character position where the insertion begins.
insert-str	string	The new string to copy into the original string at the character position.

Return Value

string
Returns the modified string.

Example

```
(make-string "The house is blue")
(string-insert! "The house is blue" 3 "ir")
                    => Their house is blue
```

string-left-trim

Syntax

```
(string-left-trim source chars)
```

Description

Removes the specified characters from the specified source string from the left end of the source.

The specified source string is left intact. The characters can be specified as a character type, a list of characters, a vector, or a string.

Parameters

Name	Type	Description
source	string	The string to trim.
chars	character, string, list, or vector	The characters to trim from the source string.

Return Value

string
Returns a new string with all of the specified characters trimmed from left.

Example

```
(string-left-trim "aa3bcde9fg" "a f g")      => "3bcde9fg"
```

string-length

Syntax

```
(string-length source)
```

Description

Returns the length of a specified string.

Parameters

Name	Type	Description
source	string	The string to measure.

Return Value

integer
The length of the specified source.

Examples

```
(string-length "abcdefg")     => 7

(string-length "12345")       => 5
```

string-length!

Syntax

```
(string-length! dest-str new-len [fill-char])
```

Description

Alters the length of the string. *dest-str* must be mutable. If lengthened, you can specify extra characters to fill the string.

Parameters

Name	Type	Description
dest-str	string	The original string to be modified.
new-len	integer	The new byte length.
fill-char	character	The characters to fill any new bytes created.

Return Value

string
Returns the modified string.

Examples

```
(define str (make-string 7 #\s))  => "sssssss"
(string-length! str 4)            => "ssss"

(define str (make-string 3 "ab")) => "ababab"
(string-length! str 8 #\7)        => "ababab77"
(string-length! str 10)           => "ababab77  "
```

string->list

Syntax

```
(string->list string)
```

Description

string->list breaks a specified string into a list of individual characters.

Parameters

Name	Type	Description
string	string	The specified string to decompose.

Return Value

list
A list composed of the individual characters making up the string.

Examples

```
(string->list "String")     => '(S t r i n g)

(string->list "17")         => '(1 7)
```

string-lrc

Syntax

```
(string-lrc string mod)
```

Description

Performs a longitudinal redundancy check by successively adding up the byte values in the specified string and performing modulo on the resulting sum. The modulo value must be a number between 1 and 255 on all bytes in a string using the lrc algorithm.

Parameters

Name	Type	Description
string	string or path	The string to check.
mod	integer	The value to use in performing modulo on the result of the lrc.

Return Value

integer
Returns the lrc of the specified string.

Examples

```
(string-lrc "AAAA" 255)     => 5

(string-lrc "AA" 100)       => 30
```

string->ntohl

Syntax

```
(string->ntohl string)
```

Description

Converts a binary blob that is a representation of a long integer in the network format (32-bit).

Parameters

Name	Type	Description
string	string	A long integer. Must be 4-byte in length.

Return Value

integer
Returns an integer.

Examples

```
(string->ntohl "aaa")  => {MONK_EXCEPTION}

(string->ntohl "aaaa") => 1633771873
```

string->ntohs

Syntax

```
(string->ntohs-> string)
```

Description

Converts a binary blob that is a representation of a short integer in the network format (16-bit).

Parameters

Name	Type	Description
string	string	A string integer with a length greater than 1.

Return Value

string
Returns a string in two-byte network byte order.

Examples

```
(string->ntohs "a")  => {MONK_EXCEPTION}

(string->ntohs "aa") => 24930
```

string-ref

Syntax

```
(string-ref source number)
```

Description

Returns the character appearing at the index position in the specified string. The index is a number that indicates the character's position from the beginning of the string, starting with 0.

Parameters

Name	Type	Description
source	string	The string to search.
number	integer	The index position of the desired character.

Return Value

character
Returns the character appearing at the index position in the specified string source.

Example

```
(string-ref "abcdefg" 3)    => #\d
```

string-right-trim

Syntax

```
(string-right-trim source chars)
```

Description

string-right-trim removes the specified characters from the specified source string from the right end of the source until it encounters a non-specified character. The specified source string is left intact. The characters can be specified as a character type, a list of characters, a vector, or a string.

Parameters

Name	Type	Description
source	string	The string to trim.
chars	character, string, list, vector	The characters to trim from the source string.

Return Value

string
Returns a new string with all of the specified characters trimmed from right.

Example

```
(string-right-trim "aa3bcde9fg" "a f g")      => "aa3bcde9"
```

string-set!

Syntax

```
(string-set! source index char)
```

Description

Replaces the character appearing at the index position in the source with the specified character. The index is a number that indicates the character's position from the beginning of the string, starting with 0.

Parameters

Name	Type	Description
source	string	The string to search.
index	integer	The index position of the character.
char	character	The replacement character.

Return Value

Unspecified.

Example

```
(define str (make-string 6 #\a)) => "aaaaaa"
(string-set! str 3 #\x)          => "aaaxaa"
```

string-substitute

Syntax

```
(string-substitute old new target)
```

Description

Searches the *target* string and replaces all instances of *old* with *new*.

Parameters

Name	Type	Description
old	string	The original string.
new	string	The replacement string.
target	string	The string to perform the substitution on.

Return Value

string
Returns a new string with substitutions performed.

Example

```
(string-substitute "Medical Doctor" "MD"
                    "John Doe, Medical Doctor")
         => "John Doe, MD"
```

string-tokens

Syntax

```
(string-tokens source char-delim)
```

Description

Creates a list of string tokens from the specified *source* using the specified *char-delim*.

Parameters

Name	Type	Description
source	string	The string to search.
char-bag	character, string, list, or vector	The characters to make into tokens.

Return Value

string

Returns a new list of string tokens delimited by *char-delim*. The original source is left unchanged.

Examples

```
(string-tokens "abcdef" #\c)              => (ab def)

(string-tokens "abcdef" '(#\c #\e #\g))   => (ab d f)
```

string-trim

Syntax

```
(string-trim source chars)
```

Description

Removes the specified characters from the source string and returns a new string.

The *chars* parameter can be either characters or characters in a string, list, or vector. This function trims the specified characters from both the left and right ends of the source until it encounters a non-specified character. The specified source string is left intact.

Parameters

Name	Type	Description
source	string	The string to trim.
chars	character, string, list, or vector	The characters to trim from the source string.

Return Value

string
Returns a new string with all of the specified characters removed from ends.

Example

```
(string-trim "aa3bcde9fg" "a 3 9 f g")    => "bcde"
```

string-type

Syntax

```
(string-type string)
```

Description

Returns the type of the specified string.

Parameters

Name	Type	Description
string	string	A string

Return Value

string

Returns one of the following encoding types:

:1Byte	:2Byte
:3Byte	:4Byte
:ASCII	:EBCDIC
:EUC	:SJIS
:UCS2	

Example

```
(define mystring "abcd")
(string-type mystring)    => :ASCII
```

string-type!

Syntax

```
(string-type! type string)
```

Description

Sets the type of the specified string and returns the modified string.

Parameters

Name	Type	Description
type	symbol	One of the following: :1Byte :2Byte :3Byte :4Byte :ASCII :EBCDIC :EUC :SJIS :UCS2
string	string	A string

Return Value

string
Returns a modified string.

Examples

```
(define mystring "abcd")
(string-type mystring)              => :ASCII

(define yourstring
  (string-type! :EBCDIC mystring))
(string-type yourstring)            => :EBCDIC
```

string-type?

Syntax

```
(string-type? type string)
```

Description

Tests whether specified string is of the specified type.

Parameters

Name	Type	Description
type	symbol	One of the following: :1Byte :2Byte :3Byte :4Byte :ASCII :EBCDIC :EUC :SJIS :UCS2
string	string	A string.

Return Value

Boolean

Returns **#t** if the string is of the specified type. Otherwise, it returns **#f**.

Examples

```
(define mystring "abcd")
(string-type? :ASCII mystring)      => #t

(string-type? :EBCDIC mystring)     => #f
```

string-upcase

Syntax

```
(string-upcase source)
```

Description

Converts alphabetic characters to upper case.

Parameters

Name	Type	Description
source	string	The string to manipulate.

Return Value

string

Returns a copy of the source with all alphabetic characters converted to upper case

Example

```
(string-upcase "A String")      => "A STRING"
```

subseq

Syntax

```
(subseq string start end)
```

Description

Creates a new string by copying a substring of an existing string .

The copy starts with the index *start* (inclusive) and the index *end* (exclusive). The offset starts from zero (0). The index *start* and *end* parameters must both be exact integers satisfying:

```
0 <= start <= end <= (string-length string)
```

Parameters

Name	Type	Description
pattern	string	Substring to test.
start	integer	Index position of the start of the pattern, inclusive.
end	integer	Index position of the end of the pattern, exclusive.

Return Value

string
Returns a newly-allocated string from the characters of string beginning with index *start* (inclusive) and index *end* (exclusive).

Examples

```
(subseq "abcdefg" 0 3)        => "abc"

(subseq "abcdefg" 1 4)        => "bcd"
```

substring-index

Syntax

```
(substring-index pattern target)
```

Description

Searches for the occurrence of a substring pattern within another string.

Parameters

Name	Type	Description
pattern	string	Pattern to search for.
target	string	String containing the pattern.

Return Value

integer

This function returns the character offset of the first occurrence of the substring pattern within the string. The offset starts from zero (0). If the substring pattern cannot be found, **#f** is returned.

Example

```
(substring-index "test" "This is a test string")   => 10
```

Chapter 7

Numerical Expressions

Numerical Expressions are used for numerical calculations and conversions. Calculation include scientific functions such as sine or tangent functions and format conversion functions dealing with big-endian and little-endian numerical data formats.

The number functions available are:

*

Syntax

```
(* number number...)
```

Description

Calculates the product of the input parameters. Accepts zero or more arguments. If no arguments are specified a value of 1 is returned.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number
Value of the product of the input argument(s).

Examples

```
(* )            => 1
(* 25)          => 25
(* -2 3)        => -6
(* -2 3 -4)     => 24
```

+

Syntax

```
(+ [number number...])
```

Description

Adds the input arguments. Accepts zero or more arguments. If you specify no input arguments, the number zero is returned.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number

Value of the sum of the input argument(s).

Examples

```
(+ )                => 0

(+ 50)              => 50

(+ 50 -100)         => -50

(+ 50 -100 200)     => 150
```

-

Syntax

```
(- number [number...])
```

Description

Subtracts the second argument from the first. If you specify only one argument this function subtracts that argument from zero. If you specify three or more arguments, this function is applied successively from left to right, with the result of the previous subtraction becoming the left argument for the next subtraction.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number
Value representing the difference of the input argument(s).

Examples

```
(- 123 )        => -123
(- -123)        => 123
(- 123 1)       => 122
(- 123 1 2)     => 120
```

/

Syntax

```
(/ number [number ...])
```

Description

Divides the first argument by the second argument.

If you specify only one argument, it divides 1 by that argument. If you specify three or more arguments, the division function is applied from left to right with the result of the previous division becoming the left argument (numerator) in the next division.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number
Value represent the quotient of the input argument(s).

Examples

```
(/ 25 )       => .04

(/ 100 50)    => 2

(/ 24 3 2)    => 4
```

<

Syntax

```
(< number number...)
```

Description

Determines whether the first argument is less than the second argument. If you specific three or more arguments, it returns **#t** if each input parameter is less than the input parameter that follows it. Otherwise, it returns **#f**

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean
Value of the comparison of all arguments.

Examples

```
(< 3 10)      =>    #t

(< 3 10 25)   =>    #t

(< 3 10 7)    =>    #f
```

=

Syntax

```
(= number number ...)
```

Description

Compares two or more numeric values to see if they are equal.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

Returns **#t** (true) if all the arguments are equal; otherwise returns **#f** (false).

Examples

```
(= 1 1 1)      =>     #t
(= 1 1 2)      =>     #f
```

<=

Syntax

```
(<= number number ...)
```

Description

Determines whether the first argument is less than or equal to the second argument. If you specific three or more arguments, it returns **#t** if each input parameter is less than or equal to the input parameter that follows it. Otherwise, it returns **#f.**

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean
Value of the comparison of the input arguments.

Examples

```
(<= 3 3)       =>    #t

(<= 3 10)      =>    #t

(<= 3 4 10)    =>    #t

(<= 3 4 1)     =>    #f

(<= 4 1)       =>    #f

(<= -17 3)     =>    #t
```

>

Syntax

```
(> number number...)
```

Description

Determines if the first argument is greater than the second argument. If you specific three or more arguments, it returns **#t** if each input argument is greater than the input argument that follows it. Otherwise, it returns **#f**

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean
Value of the comparison of all input arguments.

Examples

```
(> 3 10)           =>    #f
(> 4 -1)           =>    #t
(> 15 4 -17 -100)  =>    #t
```

>=

Syntax

```
(>= number number...)
```

Description

Determines whether the first argument is greater than or equal to the second argument. If you specific three or more arguments, it returns **#t** if each input parameter is greater than or equal to the input parameter that follows it. Otherwise, it returns **#f**

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

This function returns **#t** if each input parameter is greater than or equal to the input parameter that follows it. Otherwise, it returns **#f**.

Examples

```
(>= 3 10)         =>     #f

(>= 100 100)      =>     #t

(>= 4 1)          =>     #t

(>= 15 4 4 -17)  =>     #t
```

abs

Syntax

```
(abs number)
```

Description

Calculates the absolute value of the input argument.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number
Absolute value of the input argument.

Examples

```
(abs -34)     =>    34
(abs +50)     =>    50
(abs 3)       =>    3
(abs -4)      =>    4
```

acos

Syntax

```
(acos number)
```

Description

Calculates the arc cosine of the input argument. The input argument must be between -1 and 1.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Number
Arc cosine in radians. A number between 0 and pi.

Examples

```
(acos -1)       =>    3.14159265358979

(acos 1)        =>    0.0

(acos 0.896)    =>    0.460118237382662

(acos -0.22)    =>    1.79261079729169
```

asin

Syntax

```
(asin number)
```

Description

Calculates the arc sine of the input argument. The input argument must be between -1 and 1.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Number

Arc sine in radians. A number between -pi/2 and pi/2.

Examples

```
(asin -1)        =>    -1.5707963267949
(asin 1)         =>     1.5707963267949
(asin 0.896)     =>     1.11067808941223
(asin -0.22)     =>    -0.221814470496794
```

atan

Syntax

```
(atan number)
```

Description

Calculates the arc tangent of the input argument.

Parameters

Name	Type	Description
number	number	Any type of number, integer, or string.

Return Value

Number

Arc tangent in radians. A number between -pi/2 and pi/2.

Examples

```
(atan -1)        =>    -0.785398163397448

(atan 1)         =>     0.785398163397448

(atan 0.896)     =>     0.730600756424333

(atan -0.22)     =>    -0.216550304976089

(atan 1000000)   =>     1.5707953267949
```

big-endian->integer

Syntax

```
(big-endian->integer string size)
```

Description

Converts a string representing an integer in big endian format to a Monk integer. *size* specifies the size of the string in bytes and is permitted to have the values 1, 2, 3 or 4.

Parameters

Name	Type	Description
string	binary string	Binary string to be converted to a number.
size	integer	An integer the size of the binary string, in bytes (1-4).

Return Value

integer

This function returns an integer representation of the big endian number.

Examples

```
(big-endian->integer "A" 1)         =>    65
(big-endian->integer "a" 1)         =>    97
(big-endian->integer "Aa" 2)        =>    16737
(big-endian->integer "y" 1)         =>    121
```

ceiling

Syntax

```
(ceiling number)
```

Description

Calculates the smallest integer which is not smaller than the input argument.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

integer
The returned number is the next higher integer value of the input argument.

Examples

```
(ceiling 34)      =>     34
(ceiling 34.4)    =>     35
(ceiling -50)     =>    -50
(ceiling -50.1)   =>    -50
(ceiling -50.6)   =>    -50
```

cos

Syntax

```
(cos radians)
```

Description

Calculates the cosine of the input argument. The input argument must be in radians.

Parameters

Name	Type	Description
radians	radians number	Any type of number or string that converts to a number.

Return Value

Number
Value of the cosine of the input argument.

Examples

```
(cos 0)               =>    1.0
(cos 1)               =>    0.54030230586814
(cos -1)              =>    -0.54030230586814
(cos (/ 3.141592 3))  =>    0.50000018867511
```

even?

Syntax

```
(even? number)
```

Description

Determines whether the input argument is an even integer.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

This function returns **#t** if the integer is even. Otherwise, it returns **#f**.

Examples

```
(even? 12)      =>    #t
(even? 12.1)    =>    #f
(even? 12.8)    =>    #f
(even? -3)      =>    #f
(even? -4)      =>    #t
(even? 1558)    =>    #t
```

exp

Syntax

```
(exp number)
```

Description

Calculates the natural exponent of the input argument.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Number

Value of the exponent of the input argument.

Examples

```
(exp 1)        =>    2.71828182845905
(exp 2)        =>    7.38905609893065
(exp 3)        =>    20.0855369231877
(exp -50.6)    =>    1.0582035967718e-22
(exp 50.6)     =>    9.44714941812713e+21
```

expt

Syntax

```
(expt number1 number2)
```

Description

Calculates the value of first argument raised to the power of the second argument. Accepts real and integer arguments. If *number1* is negative, then *number2* must be an integer.

Parameters

Name	Type	Description
number1	number	Any type of number or string that converts to a number.
number2	number	Any type of number or string that converts to a number.

Return Value

number
This function returns a number.

exception
If the first argument is negative and the second argument is a real argument, an exception is returned.

Examples

```
(expt 1 1)         =>    1.0
(expt 2 2)         =>    4.0
(expt 3 3)         =>    27.0
(expt 3 3.1)       =>    30.1353256989154
(expt 3 -4.7)      =>    0.00572176613298728
(expt -5.6 2.0)    =>    {MONKEXCEPT:0001}
(expt -5.6 2)      =>    31.36
```

floor

Syntax

```
(floor number)
```

Description

Determines the greatest integer which not greater than the input argument.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

integer
The returned value is the previous lower integer value of the input argument.

Examples

```
(floor 34)        =>     34
(floor 34.4)      =>     34
(floor -50)       =>    -50
(floor -50.1)     =>    -51
(floor -50.6)     =>    -51
```

gcd

Syntax

```
(gcd number1 number2)
```

Description

Calculates the greatest common divisor of the input arguments. Each input argument must be an integer. Accepts negative values for either input argument.

Parameters

Name	Type	Description
number1	integer	Any type of number or string that converts to a number.
number2	integer	Any type of number or string that converts to a number.

Return Value

integer
Value of the greatest common divisor of the input arguments.

Examples

```
(gcd 32 -36)      =>    4

(gcd -32 +36)     =>    4

(gcd 10 -6)       =>    2

(gcd 4 5)         =>    1
```

integer?

Syntax

```
(integer? number)
```

Description

Determines whether the input argument is an integer.

Parameters

Name	Type	Description
number	any	Any type of number or string that converts to a number.

Return Value

Boolean

This function returns **#t** if the input argument is an integer. Otherwise, it returns **#f**.

Examples

```
(integer? 32)        =>    #t
(integer? -10)       =>    #t
(integer? +10)       =>    #t
(integer? -10.3)     =>    #f
(integer? "abc")     =>    #f
```

integer->big-endian

Syntax

```
(integer->big-endian number size)
```

Description

Converts an integer into a number represented as a big-endian. Takes an integer argument as its input number along with a second numeric argument that specifies the size of the big endian number to be created (1-4 bytes).

Parameters

Name	Type	Description
number	integer	An integer to convert.
size	number	The size of the integer to convert, in bytes (1-4).

Return Value

string

This function returns a string that has been formulated in big endian notation to represent the input argument to the function.

Examples

```
(integer->big-endian 65 2)       =>     A

(integer->big-endian 97 2)       =>     a

(integer->big-endian 16737 4)    =>    Aa

(integer-big-endian 121 2)       =>     y
```

integer->little-endian

Syntax

```
(integer->little-endian number size)
```

Description

Converts an integer into a number represented as a little-endian. Takes an integer argument as its input number along with a second numeric argument that specifies the size of the big endian number to be created (1-4 bytes).

Parameters

Name	Type	Description
number	integer	An integer to convert.
size	number	The size of the integer to convert, in bytes (1-4).

Return Value

string

This function returns a string that has been formulated in little endian notation to represent the input argument to the function.

Examples

```
(integer->little-endian 65 2)      =>    A

(integer->little endian 97 2)      =>    a

(integer->little endian 24897 4)   =>    aA

(integer->little endian 121 2)     =>    y
```

lcm

Syntax

```
(lcm number1 number2)
```

Description

Calculates the least common multiple of the input arguments. Each input argument must be an integer. Accepts negative values for either input argument.

Parameters

Name	Type	Description
number1	integer	Any type of number or string that converts to a number.
number2	integer	Any type of number or string that converts to a number.

Return Value

integer
This function returns an integer.

Examples

```
(lcm 12 4)      =>    12
(lcm 12 20)     =>    60
(lcm 1 10)      =>    10
(lcm 32 36)     =>    288
(lcm 32 -36)    =>    288
```

little-endian->integer

Syntax

```
(little-endian->integer string size)
```

Description

Convert a little-endian number into a Monk integer.

The little-endian number is represented as a character string, up to four bytes long. *size* specifies the size of the string (1-4 bytes)

Parameters

Name	Type	Description
string	string	Binary string to be converted to a number.
size	number	The size of the integer to convert, in bytes (1-4).

Return Value

integer
This function returns an integer.

Examples

```
(little-endian->integer "A" 1)   =>   65

(little-endian->integer "a" 1)   =>   97

(little-endian->integer "Aa" 2)  =>   24897

(little-endian->integer "y" 1)   =>   121
```

log

Syntax

```
(log number)
```

Description

Calculates the natural logarithm of the input argument. Input argument must be greater than zero.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

logarithm

This function returns the natural logarithm of the input argument.

Examples

```
(log 45)        =>    3.80666248977032
(log 1.23)      =>    0.207014169384326
(log 100000)    =>    11.5129254649702
(log 0)         =>    {MONKEXCEPT:0007}
```

max

Syntax

```
(max number [number...])
```

Description

Finds the maximum value of all input arguments.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number

The maximum value of the input parameters.

Examples

```
(max 10)              =>     10
(max 10 -2)           =>     10
(max 10 -2 10.1)      =>     10.1
(max -1000 -2000)     =>     -1000
```

min

Syntax

```
(min number [number ...])
```

Description

Finds the minimum value of all input arguments.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number
The minimum value of the input parameters.

Examples

```
(max 10)              =>    10
(max 10 -2)           =>    -2
(max 10 -2 10.1)      =>    10
(max -1000 -2000)     =>    -2000
```

modulo

Syntax

```
(modulo number modulus)
```

Description

Calculates the value of *number* reduced by *modulus*. Both arguments must be integer and the second argument must be non-zero. If *modulus* is positive, then the result is the positive or zero remainder when *number* is divided by *modulus*. If *modulus* is negative, then the result is the negative or zero remainder when *number* is divided by *modulus.*

Parameters

Name	Type	Description
number1	integer	Must be an integer.
number2	integer	Must be an integer.

Return Value

integer
Value of the modulo of the division of the two input arguments.

Examples

```
(modulo 17 7)      =>    3

(modulo 18 7)      =>    4

(modulo 19 7)      =>    5

(modulo -19 7)     =>    2

(modulo 19 -7)     =>    -2

(modulo -19 -7)    =>    -5
```

negative?

Syntax

```
(negative? number)
```

Description

Determines whether the input argument is a negative number.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

This function returns **#t** if the input argument is a negative number. Otherwise, it returns **#f**.

Examples

```
(negative? 2)      =>     #f

(negative? 2.1)    =>     #f

(negative? -3)     =>     #t

(negative? -3.6)   =>     #t
```

number?

Syntax

```
(number? number)
```

Description

Determines whether the input argument is a number.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

This function returns **#t** if the input argument is a number. Otherwise, it returns **#f**.

Examples

```
(number? 32)      =>    #t
(number? -10)     =>    #t
(number? +10)     =>    #t
(number? 'a)      =>    #f
(number? "abc")   =>    #f
(number? #\a)     =>    #f
```

number->integer

Syntax

```
(number->integer number)
```

Description

Translates a number into the corresponding integer. If the number has a fractional part, the fractional part is truncated (removed and no rounding performed).

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

integer
Returns the integer corresponding to the input number.

Examples

```
(number->integer 65)       =>    65

(number->integer -40)      =>    -40

(number->integer 3.99)     =>    3

(number->integer "Hello")  =>    {MONK_EXCEPTION}
```

number->real

Syntax

```
(number->real number)
```

Description

Translates a number into the corresponding real number data type.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

real number
Returns the real number corresponding to the input number.

Examples

```
(number->real 65)       =>    65

(number->real -40)      =>    -40

(number->real 3.99)     =>    3.99

(number->real "Hello")  =>    {MONK_EXCEPTION}
```

number->uint

Syntax

```
(number->uint number)
```

Description

Converts a number into the corresponding unsigned integer. The bits for the input number become the bits for the unsigned integer—no interpretation is done.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

uint
Returns the unsigned integer corresponding to the input number.

Examples

```
(number->uint 65)       =>    65

(number->uint -40)      =>    40

(number->uint 3.14)     =>    3

(number->uint "Hello")  =>    {MONK_EXCEPTION}
```

odd?

Syntax

```
(odd? number)
```

Description

Determines whether the input argument is an odd number.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

This function returns **#t** if the input argument is an odd number. Otherwise, it returns **#f**.

Examples

```
(odd? 23)      =>    #t

(odd? -40)     =>    #f

(odd? 20)      =>    #f

(odd? 12.3)    =>    #f
```

positive?

Syntax

```
(positive? number)
```

Description

Determines whether the input argument is a positive number. Zero is considered a positive number.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

This function returns **#t** if the input argument is a positive number. Otherwise, it returns **#f**.

Examples

```
(positive? 2)       =>     #t

(positive? -3.3)    =>     #f

(positive? 0)       =>     #t
```

quotient

Syntax

```
(quotient number1 number2)
```

Description

Divides *number1* by *number2* ignoring the remainder.

Parameters

Name	Type	Description
number1	integer	Must be an integer.
number2	integer	Must be an integer.

Return Value

integer
Value of the integer portion of the quotient.

Examples

```
(quotient 22 3)      =>    7
(quotient 21 3)      =>    7
(quotient 20 3)      =>    6
(quotient 20 -3)     =>    -6
(quotient -20 -3)    =>    6
(quotient -20 3)     =>    -6
```

real?

Syntax

```
(real? number)
```

Description

Determines whether the input argument is a real number or converts to a real number. Integers are considered real.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

Value of **#t** if the input argument is a real number. Otherwise, it returns **#f**.

Examples

```
(real? 32)          =>     #t
(real? -10)         =>     #t
(real? -10.3)       =>     #t
(real? 0)           =>     #t
(real? 'a)          =>     #f
(real? "abc")       =>     #f
(real? "123.456")   =>     #t
```

remainder

Syntax

```
(remainder number1 number2)
```

Description

Takes the input arguments and divides *number1* by *number2* to determine the remainder.

Parameters

Name	Type	Description
number1	integer	Must be an integer.
number2	integer	Must be an integer.

Return Value

integer
This function returns the remainder of the integer division of the two numbers input to the function.

Examples

```
(remainder 10 4)          =>    2

(remainder 100 25)        =>    0

(remainder 3 5)           =>    3

(remainder -1000 -2000)   =>    -1000

(remainder "12" "5")      =>    2
```

round

Syntax

```
(round number)
```

Description

Rounds the input argument to the nearest integer.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number

This function returns the rounded value of the number input to the function.

Examples

```
(round 34)       =>    34
(round 34.4)     =>    34
(round 34.5)     =>    35
(round -50.1)    =>    -50
(round -50.5)    =>    -51
(round 0)        =>    0
```

sin

Syntax

```
(sin radians)
```

Description

Calculates the sine of the input argument. The input arguments is expressed in radians.

Parameters

Name	Type	Description
radians	number	Any type of number or string that converts to a number.

Return Value

number
Value of the sine of the input argument.

Examples

```
(sin 1)                 =>    0.841470984807897
(sin 0.896)             =>    0.78083420977798
(sin (/ 3.1415926 2))   =>    1.0
```

sqrt

Syntax

```
(sqrt number)
```

Description

Calculates the square root of the input argument. The input argument must be non-negative.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

number
This function returns a real number. If a negative number is entered, an exception is returned.

Examples

```
(sqrt 0)        =>    0.0
(sqrt 1)        =>    1.0
(sqrt 9)        =>    3
(sqrt 90)       =>    9.48683298050514
(sqrt -180)     =>    {MONKEXCEPT:0053}
```

tan

Syntax

```
(tan radians)
```

Description

Calculates the tangent of the input argument. The input argument is expressed in radians.

Parameters

Name	Type	Description
radians	number	Any type of number or string that converts to a number.

Return Value

number
Value of the tangent of the input argument.

Examples

```
(tan -1)                =>    -1.5574077246549
(tan 1)                 =>    1.5574077246549
(tan 0.896)             =>    1.24985808686053
(tan (/ 3.1415926 4))   =>    1
(tan 10)                =>    0.648360827459087
```

truncate

Syntax

```
(truncate number)
```

Description

Removes the decimal point and any numbers following the decimal point.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

integer
Value of the integer portion of the input argument.

Examples

```
(truncate 34)       =>    34

(truncate 50.1)     =>    50

(truncate .123)     =>    0

(truncate -80.9)    =>    -80
```

uint?

Syntax

```
(uint? number)
```

Description

Checks to see if the input number is an unsigned integer. For purposes of this function an integer and an unsigned integer are not the same.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

Returns **#t** (true) if the number is an unsigned integer; otherwise returns **#f** (false).

Examples

```
(uint? (number->uint 65))      =>    #t
(uint? 65)                     =>    #f
(uint? -40)                    =>    #f
(uint? 3.99)                   =>    #f
(uint? "Hello")                =>    #f
```

zero?

Syntax

```
(zero? number)
```

Description

Determines whether the input argument is zero.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.

Return Value

Boolean

This function returns **#t** if the input argument is zero. Otherwise, it returns **#f**.

Examples

```
(zero? 32)        =>     #f
(zero? -10)       =>     #f
(zero? 10.3)      =>     #f
(zero? 0)         =>     #t
(zero? -0)        =>     #t
(zero? "0.0")     =>     #t
```

Chapter 8

Pairs and Lists

A pair is a structured data type having two parts, called the **car** and the **cdr**. A pair is indicated by enclosing the car and the cdr in parentheses and separating them by a period with whitespace on either side. For example, the expression (a . b) is a pair where the car is a and the cdr is b. Note that (a.b) is not proper notation for a pair.

A list is defined recursively as either an empty list, indicated by (), or a pair whose cdr is another list. For example, (a . ()) is a list since the cdr is the empty list.

All non-empty lists are pairs by definition. But not all pairs are lists since the cdr of a pair could be something other than a list.

Example 1: The expression (a . b) is a pair but not a list since the cdr, b, is neither a list nor an empty list.

Example 2: The expression (a . (b)) is both a pair and a list, since the cdr, (b) is a list having a single element. The equivalent expression for (b) is (b . ()) making it clear that (b) is a list by the recursive definition. An equivalent expression for (a . (b)) is (a . (b . ())).

Example 3: The expression (a b c) is both a pair and a list, because all non-empty lists are pairs. The notation (a b c) is shorthand for the equivalent expression (a . (b . (c . ()))).

Lists cannot be modified if constants.

Lists, like vectors, are passed by reference and are accessed differently than other Monk variable types. For more information on how these are accessed, see **"Vectors in Monk" on page 226**.

Functions which operate on lists are shown here:

list on page 214

list? on page 215

set-cdr! on page 225

append

Syntax

```
(append arg1 arg2...)
```

Description

Creates a new list by appending *arg2*, ... to *arg1*.

Arg1 must be a list. *Arg2* may be any expression. If *arg2* is a list, then append returns a proper list. If *arg2* is any other type, then append returns an improper list. If *arg1* is an empty list, then append returns *arg2* as the result.

Parameters

Name	Type	Description
arg1	list	The primary object. This argument must be a list.
arg2	any	The object to append.

Return Value

list
If *arg2* is a list, append returns a list.

pair
If *arg2* is any other type, append returns an improper list.

type
If *arg1* is an empty list, append returns *arg2* as the result.

Examples

```
; appends two lists and returns a list
(append '(a b) '(c d)          => (a b c d)

; appends empty list and symbol and returns a symbol
(append '( ) (a))              => a

; appends a list and a symbol and returns a dotted pair
(append '(c d) 'a)             => (c . (d . a))
```

assoc

Syntax

```
(assoc key alist)
```

Description

Tests each pair in the association list until it finds a pair whose car is equivalent to the object. It returns the pair if found. Otherwise, returns **#f.** The **assoc** function uses the procedure **equal?** to perform the test.

Parameters

Name	Type	Description
key	any	The object to search for.
alist	list	The association list to search.

Return Value

pair
The pair whose car is equivalent to the key.

Boolean
If the key was not found, **#f** is returned.

Examples

```
(define e '((a 1)(b 2)(c 3)))

(assoc 'a e)                        =>    (a 1)

(assoc 'b e)                        =>    (b 2)

(assoc 'd e)                        =>    #f

(assoc (list 'a)'(((a))((b))((c)))  =>    ((a))
```

assq

Syntax

```
(assq key alist)
```

Description

Tests each pair in the association list until it finds a pair whose car is equivalent to the key. It returns the pair if found. Otherwise, returns **#f**. This function uses the procedure **eq?** to perform the test.

Parameters

Name	Type	Description
key	any	The object to search for.
alist	list	The association list to search.

Return Value

pair
The pair whose car is equivalent to the key.

Boolean
If the key was not found, **#f** is returned.

Examples

```
(define e '((a 1)(b 2)(c 3)))

(assq 'a e)                          =>     (a 1)

(assq 'b e)                          =>     (b 2)

(assq 'd e)                          =>     #f

(assq (list 'a)'(((a))((b))((c))))   =>     ((a))

(assq 5 '((2 3)(5 7)(11 13)))        =>     (5 7)
```

assv

Syntax

```
(assv key alist)
```

Description

Tests each pair in the association list until it finds a pair whose car is equivalent to the key. This function uses the procedure **eqv?** to perform the test.

Parameters

Name	Type	Description
key	any	The object to search for.
alist	list	The association list to search.

Return Value

pair
The pair whose car is equivalent to the key.

Boolean
If the key was not found, **#f** is returned.

Examples

```
(define e '((a 1)(b 2)(c 3)))
(assv 'a e)                             =>     (a 1)

(assv 'b e)                             =>     (b 2)

(assv 'd e)                             =>     #f

(assv (list 'a)'(((a))((b))((c))))      =>     ((a))

(assv 5 '((2 3)(5 7)(11 13)))           =>     (5 7)
```

car

Syntax

```
(car pair)
```

Description

Returns the car of a pair.

Parameters

Name	Type	Description
pair	pair	The pair or list to test.

Return Value

car
The car of the given pair or list.

Examples

```
(car '(a b c d))      =>    a
(car '(1 . 2))        =>    1
(car '( ))            =>    {MONK_EXCEPTION}
```

cdr

Syntax

```
(cdr pair)
```

Description

Returns the cdr of a pair.

Parameters

Name	Type	Description
pair	pair	The pair.

Return Value

cdr
The contents of the cdr field.

Examples

```
(cdr '(a b c d))      =>    b c d
(cdr '(1 . 2))        =>    2
(cdr '( ))            =>    {MONK_EXCEPTION}
```

caar...cddddr

Syntax

```
(caar...cddddr)
```

Description

Returns the car, the cdr or the successive combinations of car and cdr.

The car and cdr of a list may each be nested up to four levels deep. There are 28 functions in this group: **caar, cadr, cdar, cddr, caaar, caadr, cadar, caddr, ..., caaar, ..., cdddr**.

Parameters

Name	Type	Description
list	list	The list.

Return Value

list
A list representing the expected nesting level.

Examples

```
(caar (cdddr '(a b c ((d e) f)) ) )   =>    (d e)

(cddddr '(a b c d e f))               =>    (e f)

(cdddddr '(a b c d e f))              =>    {MONK_EXCEPTION}
```

cons

Syntax

```
(cons obj1 obj2)
```

Description

Creates a new pair having *obj1* as its car and *obj2* as its cdr.

Parameters

Name	Type	Description
obj1	any	Any object. Becomes the car of the pair
obj2	any	Any object. Becomes the cdr of the pair

Return Value

pair
The pair whose car is *obj1* and whose cdr is *obj2*.

Examples

```
(cons 'a '())            =>    (a)

(cons '(a) '(b c d) )    =>    ((a) b c d )

(cons 'a  3)             =>    (a . 3)

(cons '(a b) ' c)        =>    ((a b) . c)
```

length

Syntax

```
(length list)
```

Description

Determines the length of a proper list.

Parameters

Name	Type	Description
list	list	The list to test.

Return Value

integer
The number of elements in the list.

Examples

```
(length '(a b c))              =>    3
(length '(a (b) (c d e f)))    =>    3
(length '())                   =>    0
```

list

Syntax

```
(list [obj1 obj2...])
```

Description

Creates a list from the given arguments.

Parameters

Name	Type	Description
obj1	any	Argument to concatenate into a list.
obj2	any	Argument to concatenate into a list.

Return Value

list
The list created from the given arguments.

Examples

```
(list 'a 'b 'c)           => (a b c)

(list 'a (+ 3 4) 'c)      => (a 7 c)

(list)                    => ( )
```

list?

Syntax

```
(list? obj)
```

Description

Determines if the given object is a proper list.

Parameters

Name	Type	Description
obj	any	The object to test if it is a list.

Return Value

Boolean

Returns **#t** if the object is a list. Otherwise, it returns **#f**.

Examples

```
(list? '(a b c))     =>    #t
(list? '( ))         =>    #t
(list? '(a . b))     =>    #f
```

list-ref

Syntax

```
(list-ref list num)
```

Description

Returns the element of a given list found at the index position indicated by the number. List indexing is zero-based. If you specify an index value equal to or greater than the number of elements in the list, an exception is raised.

Name	Type	Description
list	list	The list to test.
num	number	The index position of the required element.

Return Value

element
Returns the element found at the index position specified by the number.

Examples

```
(list-ref '(a b c d) 0)     =>    a

(list-ref '(a b c d) 2)     =>    c

(list-ref '(a b c d) 4)     =>    {MONKEXCEPT:0102}
```

list-tail

Syntax

```
(list-tail list num)
```

Description

Creates a sublist obtained of those elements of a given list remaining after omitting the first number of elements. If you specify a number greater than the number of elements in the list, an exception will be raised.

Parameters

Name	Type	Description
list	list	The list to test.
num	number	The number of elements to ignore when determining the sublist.

Return Value

list
List values created by deleting the initial elements.

Examples

```
(list-tail '(a b c d) 2)    =>     (c d)

(list-tail '(a b c d) 4)    =>     ()

(list-tail '(a b c d) 5)    =>     {MONKEXCEPT:0102}
```

member

Syntax

```
(member obj list)
```

Description

Creates a sublist representing the cdr of the given list whose car is the specified object. If the object does not occur in the list, then **member** returns **#f**. **member** uses the function **equal?** to perform the test between the object and the list.

Parameters

Name	Type	Description
obj	expression/ object	The object to search for.
list	list	The list to search for the object.

Return Value

sublist
Those elements of the list whose car satisfies **equal?** to the object.

Boolean
If the object was not found, **#f** is returned.

Examples

```
(member 'a '(a b c))                 =>     (a b c)

(member 'b '(a b c))                 =>     (b c)

(member 'a '(b c d))                 =>     #f

(member (list 'a) '( b (a) c ))      =>     ((a) c)
```

memq

Syntax

```
(memq obj list)
```

Description

Creates a sublist representing the cdr of the given list whose car is the specified object. If the object does not occur in the list, then **memq** returns **#f**. **memq** uses the function **eq?** to perform the test between the object and the list.

Parameters

Name	Type	Description
obj	expression/ object	The object to search for.
list	list	The list to search for the object.

Return Value

sublist
Those elements of the list whose car satisfies **equal?** to the object.

boolean
If the object was not found, **#f** is returned.

Examples

```
(memq 'a '(a b c))              =>    (a b c)

(memq 'b '(a b c))              =>    (b c)

(memq 'a '(b c d))              =>    #f

(memq (list 'a)  '( b (a) c))   =>    #f

(memq 101 '(100 101 102))       =>    (101 102)
```

memv

Syntax

```
(memv obj alist)
```

Description

Creates a sublist representing the cdr of the given list whose car is the specified object. If the object does not occur in the list, then **memv** returns **#f**. **memv** uses the function **eqv?** to perform the test between the object and the list.

Parameters

Name	Type	Description
obj	expression/ object	The object to search for.
list	list	The list to search for the object.

Return Value

sublist
Those elements of the list whose car satisfies **equal?** to the object.

Boolean
If the object was not found, **#f** is returned.

Examples

```
(memv 'a '(a b c))                =>    (a b c)

(memv 'b '(a b c))                =>    (b c)

(memv 'a '(b c d))                =>    #f

(memv (list 'a) '(b (a) c))       =>    #f

(memv 101 '(100 101 102))         =>    (101 102)
```

null?

Syntax

```
(null? obj)
```

Description

Determines if the argument is an empty list.

Parameters

Name	Type	Description
obj	any	The object to test.

Return Value

Boolean

Returns **#t** if the object is an empty list. Otherwise, it returns **#f**.

Examples

```
(null? '(a b c))        =>    #f

(null? '())             =>    #t

(null? '(a . b)         =>    #f
```

pair?

Syntax

```
(pair? obj)
```

Description

Determines if the argument is a pair.

Parameters

Name	Type	Description
obj	expression	The object to test.

Return Value

Boolean
Returns **#t** if the object is a pair. Otherwise, it returns **#f**.

Examples

```
(pair? '(a b c))        =>    #f
(pair? '())             =>    #f
(pair? '(a . b))        =>    #t
```

reverse

Syntax

```
(reverse list)
```

Description

Creates a newly allocated list consisting of the elements of the list in reverse order.

Parameters

Name	Type	Description
list	list	The list to reverse.

Return Value

list

Returns a newly allocated list consisting of the elements of the list in reverse order.

Examples

```
(reverse '(a b c))                  => (c b a)

(reverse '(a (b c) d (e (f))))     => ((e (f)) d (b c) a)
```

set-car!

Syntax

```
(set-car! pair obj)
```

Description

Stores the object into the car field of the given pair.

Parameters

Name	Type	Description
pair	pair	The pair to manipulate.
obj	expression	The object to store in the car field of the pair.

Return Value

obj
Returns an object.

Examples

```
(define f (list 1 2 3 4 5))
(set-car! f 3)
(display f)         =>    (3 2 3 4 5)

(define g (list "abc" "def"))
(set-car! g 3)
(display g)         =>    (3 "def")
```

set-cdr!

Syntax

```
(set-cdr! pair obj)
```

Description

Stores the object into the cdr field of the given pair.

Parameters

Name	Type	Description
pair	pair	The pair to manipulate.
obj	expression	The object to store in the cdr field of the pair.

Return Value

obj
Returns an object.

Examples

```
(define f (list 1 2 3 4 5))
(set-cdr! f 8)
(display f)            =>    (1 . 8)
```

A list is a pair where the cdr is another list, or the empty list. In this example, the cdr is the list (2 3 4 5) which gets replaced by (8).

```
(define g (list "abc" "def"))
(set-cdr! g 3)
(display g)            =>    (abc . 3)
```

Chapter 9

Vector Expressions

9.1 Vectors in Monk

A vector is defined as a series of elements that can be indexed by integers. A vector is indicated by enclosing the elements in #(). For example, the representation of a vector of three elements a, b, and c is #(a b c). Vectors and Lists are not the same and should not be confused.

Vectors cannot be modified if they are specified as constants. Such vectors are called *immutable*. To create a *mutable* vector use the **list->vector** or **make-vector** function.

Vectors are passed by reference and are accessed differently than other Monk variable types. Vectors are comprised of two types of arguments, immediate and non-immediate. Therefore, the vector definition will not contain any specific values for non-immediate arguments. Instead, it will contain a reference to each non-immediate argument value.

For example, two arguments are defined:

```
(define str "abc")
(define num 2)
```

Each time it is called, the vector will retrieve the value for **str**, which is a non-immediate argument type, from the **str** argument itself. However, the vector will store and always use the value originally defined for **num**, which is an immediate argument type.

If the user later changes the **str** arguments to `(define str "def")`, the vector will subsequently retrieve the new value (`def`) when it is called. However, if the user later changes the **num** argument to `(define num 3)`, the vector will still use the original value (**2**) for **num** each time it is called.

Argument types are defined as follows:

Immediate Types	Non-Immediate Types
▪ character	▪ string
▪ number	▪ vector
▪ boolean	▪ list
▪ symbol	▪ interface
▪ keyword	▪ port
	▪ function
	▪ event
	▪ time

9.2 The Vector Functions

This chapter describes the functions that are used to work with vectors. The Monk vector functions available are listed below:

list->vector

Syntax

```
(list->vector list)
```

Description

Creates a vector from a given list of elements.

Parameters

Name	Type	Description
list	list	The list of elements from which to create the vector.

Return Value

vector
The vector created from the given list of elements.

Example

```
(list->vector '(dididit dah))    => #(dididit dah)
```

make-vector

Syntax

```
(make-vector num [fill])
```

Description

Creates a vector having the specified number of elements. If the fill argument is given, each element of the created vector will be initialized to that value.

Parameters

Name	Type	Description
num	integer	The number of elements to be created in the vector.
fill	any	Optional. If specified, each element of the created vector will be initialized to this value.

Return Value

vector

A vector is created having the specified number of elements and initialized, if specified, to the given value.

Examples

```
(make-vector 2)       => #({UNSPECIFIED} {UNSPECIFIED})

(make-vector 4 4.0)   => #(4.0 4.0 4.0 4.0)
```

vector

Syntax

```
(vector obj)...)
```

Description

Creates a vector from one or more given objects.

Parameters

Name	Type	Description
obj	any	One or more objects of any data type used to create a vector.

Return Value

vector

A vector is created from the given objects.

Examples

```
(vector 'a 'b 'c)           => #(a b c)

(vector 'a (+ 3 4)  'c)     => #(a 7 c)
```

vector?

Syntax

```
(vector? obj)
```

Description

Tests if the given object is a vector.

Parameters

Name	Type	Description
obj	any	The object to test.

Return Value

Boolean

Returns **#t** if the object is a vector. Otherwise, it returns **#f**.

Examples

```
(vector?  #(a b c))      => #t

(vector? '(a b c))       => #f
```

vector->list

Syntax

```
(vector->list vector)
```

Description

Creates a list from a given vector.

Parameters

Name	Type	Description
vector	vector	The vector of elements from which to create the list.

Return Value

list

A list is created from the given vector.

Example

```
(vector->list #(dididit dah))      => (dididit dah)
```

vector-fill!

Syntax

```
(vector-fill! vector fill)
```

Description

Stores the fill value in every element of the specified vector. The specified item must be a vector.

Parameters

Name	Type	Description
vector	vector	The vector whose elements need to be filled.
fill	any	The value with which to fill the elements of the specified vector.

Return Value

vector
A vector with each element filled with the specified value.

Example

```
(vector-fill! #(a b c) 4.0)      => #(4.0 4.0 4.0)
```

vector-length

Syntax

```
(vector-length vector)
```

Description

Returns the length of the specified vector.

Parameters

Name	Type	Description
vector	vector	The vector to test.

Return Value

integer
The number of elements in the specified vector.

Example

```
(vector-length #(a b c d e))      => 5
```

vector-ref

Syntax

```
(vector-ref vector num)
```

Description

Returns the element of the specified vector whose index position corresponds to the specified number. The offset begins with 0.

Parameters

Name	Type	Description
vector	vector	The vector to manipulate.
num	integer	The index position of the vector element to return.

Return Value

element
The vector element found at the specified index position.

Examples

```
(vector-ref #(1 1 2 3 5 8 13 21) 6)    => 13

(vector-ref #(1 1 2 3 5 8 13) 8)       => {MONK_EXCEPTION}
```

vector-set!

Syntax

```
(vector-set vector num obj)
```

Description

Stores the object at the index position in the specified vector. The offset begins with 0.

Parameters

Name	Type	Description
vector	vector	The vector to manipulate.
num	integer	The index position where to store the object.
obj	any	The object to store at the index position in the vector.

Return Value

Unspecified.

Examples

```
(vector-set! #(a b c d) 3 5)     => #(a b c 5)

(vector-set! '(a b c d) 3 5)     => {MONK_EXCEPTION}
```

vector->string

Syntax

```
(vector->string vector)
```

Description

Converts the specified vector to a string.

Parameters

Name	Type	Description
vector	vector	The vector to convert.

Return Value

string
Returns a string.

Example

```
(vector->string '#(a b c))     => "abc"
```

Chapter 10

Equivalence Testing

A *predicate* is a procedure that always returns a boolean value (#t or #f). An *equivalence predicate* is a computational analogue of a mathematical equivalence relation.

In Monk, equivalence relationships exist at different levels.

Two Monk objects may be equivalent because *they are the same object* or should be regarded as the same object. The function **eqv?** on page 242 tests for this kind of equivalence. Because variables are bound to locations in memory, testing for equivalence in the **eqv?** sense may be a simple matter of comparing the address of two memory locations. No effort to compare the *contents* of memory locations need be made if the addresses already match.

Alternatively, two objects may be considered equivalent because *their contents are the same*. The function **equal?** on page 241 tests for this kind of equivalence. It may take more time to compare for equivalence in the **eqv?** sense since such comparison must examine the contents of all memory locations associated with the objects. Objects which are not equivalent in the **eqv?** sense, may be equivalent in the **equal?** sense.

Finally, two objects may be considered equivalent because *they print the same*. The function **eq?** on page 239 tests for this kind of equivalence.

eq?

Syntax

```
(eq? obj1 obj2)
```

Description

Determines if *obj1* and *obj2* should normally be regarded as the same object, except for its behavior on numbers. (Compare **eqv?**)

eq? and **eqv?** are guaranteed to have the same behavior on symbols, booleans, the empty list, pairs, procedures, non-empty strings, and vectors. **eq?**'s behavior on numbers and characters will always return either true or false, and will return true only when **eqv?** would also return true. **eq?** may also behave differently from **eqv?** on empty vectors and empty strings.

Parameters

Name	Type	Description
obj1	expression	The object to test against.
obj2	expression	The object to test for equivalence.

Return Value

Boolean

Returns **#f** if *obj2* is not equivalent of *obj1*. Otherwise, returns **#t** .

Examples

```
(eq? 'a 'a)                              => #t
(eq? '(a) '(a))                          => #f
(eq? (list 'a) (list 'a))                => #f
(eq? "a" "a")                            => #f
(eq? "" "")                              => #f
(eq? '() '())                            => #t
(eq? 2 2)                                => #t
(eq? #\A #\A)                            => #t
(eq? car car)                            => #t
(let ((n (+ 2 3))) (eq? n n))            => #t
(let ((x '(a))) (eq? x x))               => #t
(let ((x '#())) (eq? x x))               => #t
(let ((p (lambda (x) x))) (eq?  p p))    => #t
```

Notes

The implementation of **eq?** is usually much more efficient than **eqv?**, for example, as a simple pointer comparison instead of as some more complicated operation. It may not be possible to compute **eqv?** of two numbers in constant time, whereas **eq?** implemented as pointer comparison will always finish in constant time. **eq?** may be used like **eqv?** in applications using procedures to implement objects with state since it obeys the same constraints as **eqv?**.

equal?

Syntax

```
(equal? obj1 obj2)
```

Description

Determines if the obj1 and obj2 are the same type and have the same contents. **equal?** performs the least discriminating checks on the two objects. To be considered **equal?**, all the objects must do is print the same.

Parameters

Name	Type	Description
obj1	expression	The object to test against.
obj2	expression	The object to test for equivalence.

Return Value

Boolean

Returns **#f** if *obj2* is not equal to or same type as *obj1*. Otherwise, returns **#t** .

Examples

```
(equal? 'a 'a)                                => #t
(equal? '(a) '(a))                            => #t
(equal? '(a (b) c) '(a (b) c))                => #t
(equal? (make-vector 5 'a) (make-vector 5 'a)) => #t
(equal? (lambda (x) x) (lambda (y) y))        => #f
(equal? (list 'a) (list 'a))                  => #t
(equal? "a" "c")                              => #f
(equal? "" "")                                => #t
(equal? '() '())                              => #t
(equal? 2 2)                                  => #t
(equal? #1A #1A)                              => #t
(equal? car car)                              => #f
(let ((n (+ 2 3))) (equal? n n))              => #t
(let ((x '(a))) (equal? x x))                 => #t
(let ((p (lambda (x) x))) (equal? p p))       => #f
```

eqv?

Syntax

```
(eqv? obj1 obj2)
```

Description

Determines if *obj1* and *obj2* should normally be regarded as the same object. (Compare to **eq?**) **eqv?** returns **#t** if:

- *obj1* and *obj2* are both **#t** or both **#f**.
- *obj1* and *obj2* are both symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2)    => #t
```

- *obj1* and *obj2* are both characters, and are the same character according to the **char=?** procedure (see **char=?** on page 79).
- both *obj1* and *obj2* are the empty list.
- *obj1* and *obj2* are pairs, vectors, or strings that denote the same location in the store.
- *obj1* and *obj2* are procedures whose location tags are equal.

The **eqv?** expression returns **#f** if:

- *obj1* and *obj2* are of different types.
- one of *obj1* and *obj2* is **#t** but the other is **#f**.
- *obj1* and *obj2* are symbols but:

```
(string=? (symbol->string obj1)
          (symbol->string obj2)) => #f
```

- *obj1* and *obj2* are numbers for which the **char=?** procedure (see **char=?** on page 79) returns **#f.**
- one of *obj1* and *obj2* is an empty list but the other is not.
- *obj1* and *obj2* are pairs, vectors, or strings that denote distinct locations.
- *obj1* and *obj2* are procedures that would behave differently (return different values or have different side effects) for some arguments.

Parameters

Name	Type	Description
obj1	any	The object to test against.
obj2	any	The object to test for equivalence.

Return Value

Boolean

Returns a **#f** if *obj2* is not equivalent of *obj1*. Otherwise, returns **#t** .

Examples

```
(eqv? 'a 'a)                              =>  #t
(eqv? 'a 'b)                              =>  #f
(eqv? 2 2)                                =>  #t
(eqv? '() '())                            =>  #t
(eqv? 10000000 10000000)                  =>  #t
(eqv? (cons 1 2) (cons 1 2))              =>  #f
(eqv? (lambda () 1) (lambda () 2))        =>  #f
(eqv? #f 'nil)                            =>  #f
(let ((p (lambda (x) x))) (eqv? p p))     =>  #t
```

Notes

The following examples illustrate cases in which the rules specified in the description do not fully specify the behavior of **eqv?**. All that can be said about such cases is that the value returned by **eqv?** must be a Boolean.

```
(eqv? "" "")                              =>  #f
(eqv? '#() '#())                          =>  #f
(eqv?  (lambda (x) x) (lambda (x) x))     =>  #f
(eqv?  (lambda (x) x) (lambda (y) y))     =>  #f
(eqv?  '(a) '(a))                         =>  #f
(eqv?  "a" "a")                           =>  #f
(eqv? (b) (cdr '(a b)))                   =>  #f
(let ((x '(a))) (eqv?  (x x)))            =>  #t
(eqv? (list 'a) (list 'a))                =>  #f
(eqv? #\A #\A)                            =>  #t
(eqv? car car)                            =>  #t
(let ((n (+ 2 3))) (eqv?  n n))           =>  #t
(let ((x '(a))) (eqv?  x x))              =>  #t
```

Chapter 11

Conversion Procedures

The numerical input and output functions include:

number->string

Syntax

```
(number->string number [radix])
```

Description

Translates a number into the string representation by the *radix*.

Radix must be one of 2, 8, 10, or 16. If you specify no *radix*, base 10 is assumed. If *number* is real, no translation is done.

Parameters

Name	Type	Description
number	number	Any type of number or string that converts to a number.
radix	number	The base value of the number (2, 8, 10, 16).

Return Value

string

This function returns the string representation of the input number in *radix*.

Examples

```
(number->string 65)      =>    65
(number->string -40)     =>    -40
(number->string 3.14)    =>    3.14
(number->string 10 8)    =>    12
(number->string 10. 8)   =>    10
```

string->number

Syntax

```
(string->number string)
```

Description

Translates a string into a number.

Parameters

Name	Type	Description
string	string	Any string that consists of numeric characters.

Return Value

number or Boolean
This function returns a number. Otherwise, it returns **#f**.

Examples

```
(string->number "123")     =>    123

(string->number "1")       =>    1

(string->number "13.4")    =>    13.4

(string->number "abc")     =>    #f
```

keyword?

Syntax

```
(keyword? string)
```

Description

Determines whether the specified string is a keyword.

Parameters

Name	Type	Description
string	string	The string to verify.

Return Value

Boolean

Returns **#t** if the specified string is a keyword; otherwise, returns **#f**.

Example

```
(keyword? "not-a-keyword")      => #f
```

string->symbol

Syntax

```
(string->symbol string)
```

Description

Creates a symbol from the specified string.

Parameters

Name	Type	Description
string	string	The string to make into a symbol.

Return Value

symbol
A symbol created from the specified string.

Examples

```
(string->symbol "mISSISSIppi")              =>    mISSISSIppi

(symbol?(string->symbol "mISSISSIppi"))     =>    #t
```

symbol->string

Syntax

```
(symbol->string symbol)
```

Description

Creates a string from the specified symbol.

Parameters

Name	Type	Description
symbol	any	The symbol to make into a string.

Return Value

string
A string created from the specified symbol.

Example

```
(symbol->string 'flying-fish)    =>    "flying-fish"
```

char->integer

Syntax

```
(char->integer char)
```

Description

Returns the ASCII integer representation of the specified character.

Parameters

Name	Type	Description
char	character	The character for translation.

Return Value

integer
The integer representation of the specified character.

Examples

```
(char->integer #\b)      => 98
(char->integer #\#)      => 35
(char->integer #\\)      => 92
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

integer->char

Syntax

```
(integer->char num)
```

Description

This function returns the character for the specified number.

Parameters

Name	Type	Description
num	integer	The number representation of a character.

Return Value

character
The character represented by the specified number.

Examples

```
(integer->char 100)     => \#d
(integer->char 50)      => \#2
(integer->char 98)      => \#b
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

Chapter 12

File I/O Expressions

Monk supports the ability to open files, read data from files and write data to files.

A Monk structured data type called a *port* is used to track the status of the file it is associated with. A port is the data type returned by the file open functions. For example to prepare a file for reading you would execute code like this:

```
(define myfileptr (open-input-file "c:\data\employee.dat"))
```

Then the variable **myfileptr** which is a port would later be used in later function calls to read from the **employee.dat** file.

The File I/O Expressions are:

clear-port-callback

Syntax

```
(clear-port-callback port)
```

Description

Clears the current callback procedure from the specified port.

Parameters

Name	Type	Description
port	port	Handle to the open file.

Return Value

Unspecified.

Example

```
(clear-port-callback port1)
```

close-port

Syntax

```
(close-port port)
```

Description

Closes the specified port, if open.

Parameters

Name	Type	Description
port	port	Handle to open port.

Return Value

Unspecified.

Example

```
(close-port fp)          =>     {MONK_UNSPECIFIED}
```

current-debug-port

Syntax

```
(current-debug-port)
```

Description

Routes the output resulting from any debug flags set to the specified port defined in monkext.monk.

Parameters

None.

Return Value

The port where the debug output is sent.

Example

```
(current-debug-port)                 =>     #{Debug-port}
```

current-error-port

Syntax

```
(current-error-port)
```

Description

Returns the current error port.

Parameters

None.

Return Value

This function returns the current error port.

Example

```
(current-error-port)      =>     #{output-port}
```

current-input-port

Syntax

```
(current-input-port)
```

Description

Returns the current standard input port.

Parameters

None.

Return Value

This function returns the standard input port.

Example

```
(current-input-port)        =>     #{Input-port}
```

current-output-port

Syntax

```
(current-ouput-port)
```

Description

Returns the current standard output port.

Parameters

None.

Return Value

This function returns the standard output port.

Example

```
(current-output-port)       =>     #{output-port}
```

current-warning-port

Syntax

```
(current-warning-port)
```

Description

Returns the current warning port.

Parameters

None.

Return Value

The port.

Example

```
(current-warning-port)        =>     #{output-port}
```

ftell

Syntax

```
(ftell port)
```

Description

Obtains the current read/write position of the port.

Parameters

Name	Type	Description
port	port	Handle to the open file.

Return Value

integer

The **ftell** function returns a positive integer (including 0) to indicate the current position of the read/write position within an open port.

If the file is not open, it will return an error.

Examples

```
(define fp (open-input-file "/home/user1/temp-text"))
(ftell fp)              => 0

(read fp 80)
(ftell fp)              => 80
```

get-port-callback

Syntax

```
(get-port-callback port)
```

Description

Retrieves the current callback procedure from the specified port.

Parameters

Name	Type	Description
port	port	Handle to the open file.

Return Value

This procedure returns the callback procedure from the specified port.

Example

```
(get-port-callback port1)
```

input-string-port?

Syntax

```
(input-string-port? port)
```

Description

Tests whether the specified port is an input string port.

Parameters

Name	Type	Description
port	port	Handle to the input string.

Return Value

Boolean

Returns **#f** if the port is not an input string port. Otherwise, it evaluates to **#t**.

Examples

```
(define fp (open-input-file "/home/user1/test.txt"))
(input-string-port? fp)                      =>    #f

(define buffer "the quick brown fox jumps over the lazy dog")
(define fp2 (open-input-string buffer))
(input-string-port? fp2)                     =>    #f
```

open-append-file

Syntax

```
(open-append-file filename [:file64])
```

Description

Opens a file in append mode.

If the file does not exist, it will be created, if possible.

Parameters

Name	Type	Description
filename	string	Full path to the file.
:file64	optional	Used to provide large file support. This option is available on the following platforms: ▪ AIX ▪ Compaq ▪ HP ▪ Linux ▪ Sun This parameter is optional.

Return Value

This function returns a port to the open file.

Example

```
(open-append-file "/home/user1/test.txt")   =>  #{Append-port}

(open-append-file "/home/user1/test.txt" :file64)   =>  #{Append-
port}
```

open-input-file

Syntax

```
(open-input-file filename [:file64])
```

Description

Opens a file in input mode.

Parameters

Name	Type	Description
filename	string	Full path to the file.
:file64	optional	Used to provide large file support. This option is available on the following platforms: ▪ AIX ▪ Compaq ▪ HP ▪ Linux ▪ Sun This parameter is optional.

Return Value

This function returns a port to the input file. If the file does not exist, it will return an error.

Example

```
(open-input-file "/home/user1/test.txt")    =>    #{Input-port}

(open-input-file "/home/user1/test.txt" :file64)    =>    #{Input-
port}
```

open-input-string

Syntax

```
(open-input-string string)
```

Description

Opens a port on the specified string in input mode.

Parameters

Name	Type	Description
string	string	Full path to the string to input.

Return Value

This function returns the port.

Example

```
(define buffer "The quick brown fox jumps over the lazy dog")
(open-input-string buffer)          =>    #{InputString-port}
```

open-output-file

Syntax

```
(open-output-file filename [:file64])
```

Description

Opens a port in output mode.

Since this function creates an output file, the directory where the file exists must be accessible and usable.

Parameters

Name	Type	Description
filename	string	Full path to the file.
:file64	optional	Used to provide large file support. This option is available on the following platforms: ▪ AIX ▪ Compaq ▪ HP ▪ Linux ▪ Sun This parameter is optional.

Return Value

This function returns the port to the output file.

Example

```
(open-output-file "output.dat")    =>    #{Output-port}

(open-output-file "output.dat" :file64)    =>    #{Output-port}
```

open-output-string

Syntax

```
(open-output-string)
```

Description

Opens a port for output.

Parameters

None.

Return Value

This function returns the port of the output string.

Example

```
(open-output-string)        =>     #{OutputString-port}
```

open-random-access-file

Syntax

```
(open-random-access-file filename [:file64])
```

Description

Opens a port in random access mode.

If the file does not exist, it will be created, if possible.

Parameters

Name	Type	Description
filename	string	Full path to the file.
:file64	optional	Used to provide large file support. This option is available on the following platforms: ▪ AIX ▪ Compaq ▪ HP ▪ Linux ▪ Sun This parameter is optional.

Return Value

Returns a port to the open file.

Example

```
(open-random-access-file "/home/user1/temp.txt")
                 => #{RandomAccess-port}

(open-random-access-file "/home/user1/
temp.txt"v:file64)
                 => #{RandomAccess-port}
```

output-port?

Syntax

```
(output-port? port)
```

Description

Tests whether the specified port is an output port type.

Parameters

Name	Type	Description
port	port	Handle to the port.

Return Value

Boolean

This function returns **#f** if the port is not an output port type. Otherwise, it evaluates to **#t**.

Example

```
(define fp4 (open-output-file "output.dat"))
(output-port? fp4)                          =>    #t
```

output-string-port?

Syntax

```
(output-string-port? port)
```

Description

Tests whether the specified port is an output string port.

Parameters

Name	Type	Description
port	port	Handle to the output string.

Return Value

Boolean

This function returns **#f** if the port is not an output string port; Otherwise, it evaluates to **#t**.

Examples

```
(define fp3 (open-outstring))
(output-string-port? fp3)                          =>    #t

(define fp4 (open-output-file "output4.dat))
(output-string-port? fp4)                          =>    #f
```

regex-string-port

Syntax

```
(regex-string-port string port from-start)
```

Description

Determines the location or index of an string on a port.

Parameters

Name	Type	Description
string	string	The string to test.
port	port	The port for input
from-start	any	Searches from start of string. If not specified, the search begins at the current position.

Return Value

Boolean
Returns **#f** if the string could not be found.

integer
Location of the string in the input file.

Example

```
(define buffer "The quick brown fox jumps over the lazy dog")
(define fp3 (open-input-string buffer))
(regex-string-port "quick" fp3)                          =>    4
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

rewind

Syntax

```
(rewind port)
```

Description

Moves an open port position to the beginning of the data.

Parameters

Name	Type	Description
port	port	Handle to the open file.

Return Value

Boolean

Returns **#f** if the rewind was not successful. Otherwise, it evaluates to **#t**.

Example

```
(rewind fp)          =>    #t
```

seek-cur

Syntax

```
(seek-cur port offset)
```

Description

Moves an open port pointer to the specified offset within the file, relative to the pointer's current position.

When the offset integer is negative, the pointer moves backward, relative to the current pointer position.

Parameters

Name	Type	Description
port	port	Handle to the open file.
offset	integer	Offset within the file.

Return Value

Boolean
Returns **#f** if the seek was not successful. Otherwise, it evaluates to **#t**.

Examples

```
(define fp (open-input-file "/home/user1/test.txt"))
(seek-set fp 72)
(display (ftel fp))           =>    72

(read fp 18)
(seek-cur fp -45)             =>    #t

(display (ftel fp))           =>    45
```

seek-set

Syntax

```
(seek-set port offset)
```

Description

Moves an open port pointer to the specified offset, relative to the beginning.

Parameters

Name	Type	Description
port	port	Handle to the open file.
offset	integer	Offset within the file.

Return Value

Boolean

Returns **#f** if the seek was not successful; Otherwise, it evaluates to **#t**.

Examples

```
(define fp (open-input-file "/home/user1/test.txt"))
(seek-set fp 18)          =>    #t

(display (ftell fp))      =>    18

(seek-set fp 30)          =>    #t

(display (ftel fp))       =>    30
```

seek-to-end

Syntax

```
(seek-to-end port)
```

Description

Moves an open port pointer to the end of the file.

Parameters

Name	Type	Description
port	port	Handle to the open file.

Return Value

Boolean

Returns **#f** if the seek was not successful. Otherwise, it evaluates to **#t**.

Example

```
(define fp (open-input-file "/home/user1/test.txt")
(seek-to-end fp)                        =>    #t
```

set-file-encoding-method

Syntax

```
(set-file-encoding-method type)
```

Description

Sets the file encoding method.

Parameters

Name	Type	Description
type	symbol	One of the following file encoding types: :1Byte :2Byte :3Byte :4Byte :ASCII :EBCDIC :UCS2 :EUC :SJIS

Return Value

Boolean

Returns **#t** (true) if a valid file encoding method is set; otherwise returns **#f** (false).

Examples

```
(set-file-encoding-method :ASCII)       =>    #t

(set-file-encoding-method :DogByte)     =>    #f

(set-file-encoding-method ASCII)        =>    {MONK_EXCEPTION}
```

set-port-callback

Syntax

```
(set-port-callback port procedure)
```

Description

Sets the callback procedure for the specified port.

Parameters

Name	Type	Description
port	port	Handle to the open file.
procedure	procedure	Callback procedure.

Return Value

Unspecified.

Example

```
(set-port-callback port1 procedure_name)
```

string-port->string

Syntax

```
(string-port->string port)
```

Description

Returns the string representing the contents of the specified port.

Parameters

Name	Type	Description
port	port	Handle to the string port.

Return Value

string
This function returns the string representing the contents of the specified port.

Example

```
(define buffer "The quick brown fox jumps over the lazy dog")

(define fp2 (open-input-string buffer))

(string-port->string fp2)
    =>    "The quick brown fox jumps over the lazy dog"
```

eof-object?

Syntax

```
(eof-object? obj)
```

Description

Tests the object as an EOF object.

Parameters

Name	Type	Description
obj	any	An object to be tested.

Return Value

Boolean

This function returns **#t** if the object is an EOF object. Otherwise, it returns **#f.**

Example

```
(define fp (open-input-file "/home/user1/test.txt")
(define eofchar (read-char fp))
(eof-object? eofchar)           =>    #f

(seek-to-end fp)
(define eofchar (read-char fp))
(eof-object? eofchar)           =>    #t
```

read

Syntax

```
(read port number)
```

Description

Reads a specified number of characters from an open port. If **read** is used to read characters from either standard input, or the port returned by (current-input-port), then **read** will not return until the specified number of characters has been read.

Parameters

Name	Type	Description
port	port	Handle to the open input/random access/string port.
number	integer	Number of characters to read.

Return Value

string
This function returns the number of characters read, or less if not available.

eof-object
The end-of-file object.

Example

```
(define fp (open-input-file "/home/user1/test.txt"))
(read fp 17)      =>    "how now brown cow"

(read (current-input-port) 15) => "This is a test."
```

Since (current-input-port) may return a port which is not a file, it cannot be known that and end-of-file type of error has occurred. Therefore, it simply waits until the 15th character can be provided.

read-char

Syntax

```
(read-char [port])
```

Description

Reads data one character at a time from an input port.

The port is optional, and if not specified, standard input is assumed. If standard input is specified, then this function will wait until a character has been entered on standard input; it will not return an end-of-file type error.

Parameters

Name	Type	Description
port	port	Handle to the open input/random access/string port.

Return Value

character
This function returns the character read from the input port.

eof-object
The end-of-file object.

Example

```
(define fp (open-input-file "/home/user1/test.txt"))
(read-char fp)                       =>    T
```

read-line

Syntax

```
(read-line port number)
```

Description

Reads characters from a port up to either the number, or end-of-line, or end of data, whichever is first.

Parameters

Name	Type	Description
port	port	Handle to the open file.
number	integer	Number of bytes to read.

Return Value

string
This function returns the specified number of bytes from an open port. If a newline is encountered before the specified number of bytes have been read, the function returns the bytes read up to, but not including the newline character.

eof-object
The end-of-file object.

Example

```
(define fp (open-input-file "/home/user1/test.txt"))
(read-line fp 80)          =>    "how now brown cow"
```

display

Syntax

```
(display object [port])
```

Description

Displays the object to the specified output port.

The port is optional. If not present, the system defaults to the standard output port.

Parameters

Name	Type	Description
object	any	The object to display at the output port.
port	port	Handle to the open port (optional).

Return Value

Unspecified.

Example

```
(define fp4 (open-output-file "output.dat"))
(display "writing to file" fp4)   =>    {MONK_UNSPECIFIED}
```

The file **output.dat** now contains:

```
writing to file
```

newline

Syntax

```
(newline [port])
```

Description

Writes a newline to the output port.

The port is optional. If not specified, the standard output port is assumed.

Parameters

Name	Type	Description
port	port	Handle to the open port (optional).

Return Value

Unspecified.

Example

```
(define fp4 (open-output-file "output.dat"))
(newline fp4)                  =>    {MONK_UNSPECIFIED}
```

The file **output.dat** now contains a newline.

write

Syntax

```
(write object size [port])
```

Description

write is similar to the **display** function except for the addition of the size parameter. **write** sends a specified number of bytes of an object to a port. If no port is specified, the bytes are sent to standard out—typically the display screen. If the number of bytes (N) to write is less than the size of the object, then only the first (N) bytes of the object are written and the rest are truncated.

Parameters

Name	Type	Description
object	any	The monk object to be written.
size	integer	The number of bytes (N) to be written
port	port	Optional. The port to which the data is written. If no port is specified, the data is sent to standard out.

Return Value

Unspecified.

Examples

```
(define fp4 (open-output-file "output.dat"))
(write "Please have a nice day." 10 fp4)  =>    {MONK_UNSPECIFIED}
```

The file **output.dat** now contains:

```
Please hav
```

write-char

Syntax

```
(write-char char port)
```

Description

Writes one character to the specified port. The port is optional. If not specified, standard output is assumed.

Parameters

Name	Type	Description
char	character	Character to write.
port	port	Handle to the open port (optional).

Return Value

Unspecified.

Example

```
(define fp4 (open-output-file "output.dat"))
(write-char #\A port2)       =>    {MONK_UNSPECIFIED}
```

The file **output.dat** now contains:

```
A
```

write-exp

Syntax

```
(write-exp obj port)
```

Description

Writes an expression to a port in a format that can be read back in by the monk engine. For example, vector objects have the output format #() and strings have double quotes around them. The port is optional. If not specified, standard output is assumed.

Parameters

Name	Type	Description
obj	any	Any valid object to write.
port	port	Handle to the open file (optional).

Return Value

Unspecified.

Examples

```
(define fp1 (open-output-file "c:\output.dat"))
(define st0 "This is ")
(define st1 "exactly what we wanted.")
(write-exp (string-append st0 st1) fp1)
```

The file **c:\output.dat** now contains:

```
"This is exactly what we wanted."
```

Note: *The quotes are included in the output because the Monk engine requires quotes around string data.*

```
(define fp1 (open-output-file "c:\output.dat"))
(define st0 #\A)
(write-exp st0 fp1)
```

The file **c:\output.dat** now contains:

```
#\A
```

Chapter 13

System Interface Functions

System Interface functions may be used to find out information *about* files that exist on the system, to load files into the Monk engine, or to execute system commands.

The System Interface functions include:

directory

Syntax

```
(directory pathstring)
```

Description

Returns the contents of the specified directory as a vector.

Parameters

Name	Type	Description
pathstring	string	The full or partial path of the directory. Will use the **load-path** value if a partial path is given. The Monk **load-path** is an internal Monk variable that is automatically set. The **load-path** is populated in part by the **SharedExe** and **SystemData** values in the **.egate.store** file. See the *e*Gate Integrator System Administration and Operations Guide* for more information about this file.

Return Value

vector

A vector of strings. The strings (vector elements) are the file names of the files and subdirectories found in the specified directory.

Boolean

Returns #f if the directory does not exist.

Examples

```
(directory "data")        =>    #(. .. ETDs FileIn.txt)

(directory "c:\test")     =>    #(. .. Doc1.txt Doc2.txt
NoMoreDocs.txt)

(directory "bogus")       =>    #f
```

file-delete

Syntax

```
(file-delete filename)
```

Description

Deletes a file.

Parameters

Name	Type	Description
filename	string	Full path to the file.

Return Value

Boolean

This function returns **#f** if the file specified does not exist or was not successfully deleted; evaluates to **#t** if the file was deleted.

Example

```
(if (file-exists? "output.dat")
   (file-delete "output.dat")
   (display "Cannot delete file: Does not exist)
)
```

file-exists?

Syntax

```
(file-exists? filename)
```

Description

Checks for the existence of a file.

Parameters

Name	Type	Description
filename	string	Full path to the file.

Return Value

Boolean

Returns **#f** if the file specified does not exist; Otherwise, it evaluates to **#t**.

Examples

```
(file-exists? "output.dat")    => #t

(file-exist? "nonfile.dat")    => #f
```

file-rename

Syntax

```
(file-rename filename1 filename2)
```

Description

Renames the original file to the new file name. You must include the full path.

Parameters

Name	Type	Description
filename1	string	The original name of the file, including the full path.
filename2	string	The new name of the file, including the full path.

Return Value

Boolean

Returns **#f** if the file specified does not exist. Otherwise, it evaluates to **#t**.

Examples

```
(file-rename
  "/home/user1/output.dat"
  "/home/user1/mytestdata.dat")       =>    #t
```

getenv

Syntax

```
(getenv variable)
```

Description

Retrieves the value of the specified environment variable.

Parameters

Name	Type	Description
variable	string	Name of the environment variable from which the value is retrieved.

Return Value

string
Returns a string representing the value of the specified environment variable.

Boolean
Returns **#f** if the variable does not exist.

Example

```
(getenv "ORACLE_HOME")    => /opt/oracle8/app/oracle/product/8.0.5
```

load

Syntax

```
(load filename)
```

Description

Reads expressions and definitions from the file specified and evaluates them sequentially.

Parameters

Name	Type	Description
filename	string	Path to the file to load. can be full or partial path by using the **load-path** setting. The Monk **load-path** is an internal Monk variable that is automatically set. The **load-path** is populated in part by the **SharedExe** and **SystemData** values in the ***.egate.store*** file. See the e**Gate Integrator System Administration and Operations Guide* for more information about this file.

Return Value

Unspecified.

Note: *If the Monk file to be loaded returns an exception, that exception will be returned by the* ***load*** *function.*

Example

```
(load "my_monk_library/my_file")    =>    {MONK-UNSPECIFIED}
```

load-directory

Syntax

```
(load-directory dirname)
```

Description

Loads all files with the .monk extension from the specified directory into the Monk environment. Performs a load on each file, ignoring all but catastrophic exceptions.

Parameters

Name	Type	Description
dirname	string	Full path to a directory.

Return Value

Unspecified.

Limitations

load-directory does not operate recursively.

For example, if a file (“loadother.monk”) is found in the directory specified by *dirname* which itself contains a **load-directory** command, the first **load-directory** command will not run to completion, but stop after the file (“loadother.monk”) is finished loading.

Example

```
(load-directory "my_monk_library")      => {MONK-UNSPECIFIED}
```

load-extension

Syntax

```
(load-extension filepath)
```

Description

Loads a shared .dll into the Monk environment.

Important: *If the specified .dll does not exist or if the filepath is too long, a severe exception condition results.*

Parameters

Name	Type	Description
filepath	string	Load path to the shared dll. Can be a partial load path. The filepath consists of the path plus the filename. The path must be 256 or fewer characters in length and the filename must be 64 or fewer characters in length, for a maximum total of 320 characters.

Return Value

Unspecified.

Example

```
(load-extension "d:/egate/client/bin/stc_dbodbc.dll")
                 =>     {MONK_UNSPECIFIED}

(load-extension "stc_monkutils.dll")
                 =>     {MONK_UNSPECIFIED}
```

putenv

Syntax

```
(putenv variable_value)
```

Description

Assigns a value to an environment variable.

Parameters

Name	Type	Description
variable_value	string	Name and value of the environment variable.

Return Value

Boolean

Returns **#f** if the operation was not successful. Otherwise, it evaluates to **#t**.

Example

```
(putenv "PROGRAM_ENV=/home/program/value")     => #t
```

system

Syntax

```
(system command [#t | :func function])
```

Description

Runs an operating system command from Monk.

The **#t** option instructs the **system** command to provide the OS return code it receives upon completion of **command** as its return value.

The **:func** ***function*** option calls the monk function specified with the OS return code as the argument.

Important: *This function must be used with extreme caution. Invoking an executable file that takes a long time to run or has the potential to hang should be avoided.*

Parameters

Name	Type	Description
command	string	The OS command to be executed.
function	symbol	A Monk function.

Return Value

Returns one of the following:

any
If no options are used, **system** returns the return value of the command being executed.

integer
If the **#t** option is used, **system** returns the OS return code.

any
If the **:func function** option is used, **system** returns the result of the Monk function specified.

Examples

The following examples use the Solaris 2.6 UNIX operating system.

```
(system "ls")           => #t

(system "ls" #t)        => 0

(system "list" #t)      => 256
```

The following example uses the Windows XP operating system.

```
(define
    myfunction
    (lambda
        (returncode)
        (display (string-append "\nOperating System Returns: "
                (number->string returncode)))))

(system "dir" :func myfunction)

                      => Operating System Returns: 0
```

Chapter 14

Standard Procedures

The Standard Procedure functions include:

14.1 Booleans

Boolean expressions are those that evaluate to either true or false. In Monk, a Boolean expression returns false **#f** or the expression is assumed to be true.

boolean? on page 301

boolean?

Syntax

```
(boolean? obj)
```

Description

Determines if the object is a Boolean value.

Parameters

Name	Type	Description
obj	expression	The object to test for being a Boolean value.

Return Value

Boolean
Value of **#t** if the argument is Boolean. Otherwise, returns false **#f**.

Examples

```
(boolean? 0)    =>    #f

(boolean? #t)   =>    #t

(boolean? #\B)  =>    #f

(boolean? #f)   =>    #t
```

14.2 Symbols

The available symbol functions are:

keyword? on page 302

symbol? on page 303

sys-procedures on page 304

sys-symbols on page 305

keyword?

Syntax

```
(keyword? string)
```

Description

Determines whether the specified string is a keyword.

Parameters

Name	Type	Description
string	string	The string to verify.

Return Value

Boolean

Returns **#t** if the specified string is a keyword; otherwise, returns **#f**.

Example

```
(keyword? "not-a-keyword")      => #f
```

symbol?

Syntax

```
(symbol? obj)
```

Description

Tests the specified object to determine if it is a symbol.

Parameters

Name	Type	Description
obj	any	The object to test.

Return Value

Boolean

Returns **#t** if the object is a symbol. Otherwise, it returns **#f**.

Examples

```
(symbol? 'foo)      =>    #t
(symbol? "bar")     =>    #f
```

sys-procedures

Syntax

```
(sys-procedures)
```

Description

Creates a list of symbols that represent the procedures defined within the current scope.

Parameters

None.

Return Value

list
Returns a list of procedures.

Example

```
(sys-procedures)

=>

($event->string $event-clear $event-parse $make-event-map $resolve-
event-definition * + - / < <= = > >= abort abs acos
and append apply asin assoc assq assv atan begin
big-endian->integer boolean? ... more functions follow ...
```

sys-symbols

Syntax

```
(sys-symbols)
```

Description

Creates a list of all the known symbols in the Monk environment.

Parameters

None.

Return Value

list
Returns a list of symbols.

Example

```
(sys-symbols)

=>

($event->string $event-clear $event-parse $make-event-map
$resolve-event-definition * + - / :1Byte :1bEUC :1bSJIS :2Byte :2bEUC
:2bSJIS :3Byte ... more symbols follow ...
```

14.3 Sequence Operators

The Sequence Operator functions include:

nth on page 306

qsort on page 307

nth

Syntax

```
(nth index/integer sequence)
```

Description

Retrieves the nth element from the specified sequence.

Parameters

Name	Type	Description
index/ integer	positive integer	The number of the element in the list to retrieve.
list	list, string, or vector	The elements of the list, enclosed in parentheses and separated by spaces.

Return Value

This function returns the contents of the nth element of the sequence.

Examples

```
(nth 3 (list "a" "b" "c" "d" "e"))          =>     d
(nth 0 ("hello" "goodbye" "red" "blue"))    =>     hello
(nth 7 "abcdefghijklmnop")                  =>     h
```

qsort

Syntax

```
(qsort list/vector procedure)
```

Description

Sorts the list or vector using the specified procedure.

Name	Type	Description
list/vector	list/vector	The list or vector to run the procedure against.
procedure	procedure	The procedure to use for comparison.

Return Value

vector or list
Can have optionally a Boolean or tri-state integer result.

Examples

```
(qsort '("b" "e" "a" "d" "c") string<=?)       => (a b c d e)

(qsort '#("zero" "bbbbb" "hello" "end") string>=?)
                    => (zero hello end bbbbb)
```

14.4 Control Features

The following are the available control functions:

apply

Syntax

```
(apply proc list)
```

Description

Calls the given procedure using the elements of the list as the arguments of that procedure.

Parameters

Name	Type	Description
proc	procedure	The procedure to be applied.
list	list	The list of elements to use as arguments to the procedure.

Return Value

result

The return from apply is the result of evaluating the procedure upon the list.

Examples

```
(apply + (list 3 4))          => 7


(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args))
    )
  )
)

((compose sqrt *) 12 75)      =>  30
```

map

Syntax

```
(map proc list1 list2...)
```

Description

Calls the given procedure using the corresponding element of each list as an argument of the procedure.

There must be as many lists as there are arguments to the procedure. If there is more than one list, all lists must be the same length.

Parameters

Name	Type	Description
proc	procedure	The procedure to apply.
list	list	The list of elements to use as arguments to the procedure.

Return Value

list

A list of results, in order. The dynamic order in which *proc* is applied to the elements of the *listN* is unspecified.

Examples

```
(map cadr '((a b) (d e) (g h)))
      => (b e h)


(map (lambda (n) (expt n n))
    '(1 2 3 4 5))
      => (1.0 4.0 27.0 256.0 3125.0)


(map + '(1 2 3) '(4 5 6)
      => (5 7 9)


(let ((count 0))
  (map (lambda () (set! count (+ count 1)) count)
       '(a b)
  )
)
      =>  (1 2)
```

procedure?

Syntax

```
(procedure? obj)
```

Description

Tests if the given object is a procedure.

Parameters

Name	Type	Description
obj	any	The object to test if it is a procedure.

Return Value

Boolean

Returns **#t** if the object is a procedure. Otherwise, it returns **#f**.

Examples

```
(procedure? car)                        => #t

(procedure? 'car)                       => #f

(procedure? (lambda (x) (* x x)))       => #t

(procedure? '(lambda (x) (* x x)))      => #f
```

14.5 Evaluation

The Evaluation function evaluates the specified object in the current environment and returns the result:

eval on page 311

eval

Syntax

```
(eval obj)
```

Description

Evaluates the specified object in the current environment and returns the result.

Parameters

Name	Type	Description
obj	any	The object to be evaluated based on the current environment.

Return Value

The result returned depends on the given object. For example, a number returns a number and a string returns a string.

Example

```
(define myfunc "display")
(display (string-append "\n The value of myfunc is: " myfunc " \n"))
((eval (string->symbol myfunc)) "\n This worked! \n")
    => This worked!
```

14.6 Literal Expressions

The literal function available is:

quote on page 312

quasiquote on page 313

Strings (""), quoted lists '(. . .), and vectors #(. . .) are immutable.

quote

Syntax

```
(quote datum)
```

or

```
'datum
```

Description

Evaluates to the object in the *datum* parameter. The *datum* can be any data type recognized by Monk. The expressions (quote *datum*) and '*datum* are equivalent in all respects. Numerical constants, string constants, character constants, and Boolean constants always evaluate to themselves, and thus they do not have to be quoted.

Parameters

Name	Type	Description
datum	expression	The object to be evaluated.

Return Value

The evaluated object.

Examples

The result is the symbol a:

```
(quote a) => a
```

The result is a non-mutable vector:

```
'#(a b c) => #(a b c)
```

quasiquote

Syntax

```
(quasiquote qqtemplate)
```

or

```
`qqtemplate
```

Description

Constructs a list or vector structure when most but not all of the desired structure is known in advance.

If no commas appear within the *qqtemplate*, the result of the evaluated (quasiquote *qqtemplate)* is equivalent to the result of evaluating (quote *qqtemplate).*

If a comma appears within the *qqtemplate*, however, the expression following the comma is evaluated ("unquoted") and its result is inserted into the structure instead of the comma and the expression.

If a comma appears immediately before an at-sign ("@"), then the following expression must evaluate to a list. The opening and closing parentheses of the list are stripped away, and the elements of the list are inserted in place of the comma and at-sign expression sequence. A comma at sign should only appear within a list or vector *qqtemplate*.

Parameters

Name	Type	Description
qqtemplate	list or vector	The structure to evaluate.

Return Value

A list or vector as the result of the evaluation of *qqtemplate*.

Examples

```
(quasiquote (list ,(+ 1 2) 4))          => (list 3 4)

`(list ,(+ 1 2) 4)                      => (list 3 4)

(let ((name 'a)) `(list ,name))         => (list a)

`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)   => (a 3 4 5 6 b)

`((foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
                        => ((foo 7) . cons)

`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)      => #(10 5 2 4 3 8)
```

Notes

Quasiquote forms can be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
                    => `(a `(b ,(+ 1 2) ,(foo , 4 d) e) f)
```

The two notations (**quasiquote** *qqtemplate*) and `(*qqtemplate*) are identical in all respects. Likewise, **(unquote *expression*)** is identical to **,(*expression*)**, and **(unquote-splicing *expression*)** is identical to **@(*expression*)**.

```
(quasiquote (list ,(+ 1 2) 4))                        => (list 3 4)

(quasiquote (a ,(+ 1 2) ,@(map abs '(4 -5 6)) b))     => (a 3 4 5 6 b)
```

Unpredictable behavior can result if any of the symbols **quasiquote**, **unquote**, or **unquote-splicing** appear in positions within a (*qqtemplate*).

14.7 Procedure

The procedure expressions, **lambda** and **lambdaq**, evaluates to a procedure:

lambda on page 315

lambdaq on page 317

lambda

Syntax

```
(lambda formals body)
```

The *formals* can have one of the following three forms:

```
(variable1 ... )
variable
(variable1 ... variableN . variableN+1)
```

Description

Creates a procedure or function. It accepts arguments (*formals*), accepts a list of expressions (*body*), and returns a procedure.

If a **lambda** expression is used in conjunction with a **define**, then the procedure which is may be executed as long as the definition remains valid. In this way, procedures and functions may be defined globally and executed as often as needed.

(variable1 ...)
fixed number of arguments, when the procedure is called the arguments will be stored in the binding of the corresponding variables.

variable
the procedure takes an unspecified number of arguments; when the procedure is called, the sequence of actual arguments are converted into a newly allocated list, and the list is stored in the binding of the variable.

(variable1 ... variableN . variableN+1)
If a space-delimited period precedes the last variable, then the procedure takes *N* or more arguments, where *N* is the number of formal arguments before the period (there must be at least one argument). The value stored in the binding of the last variable (the variable after the period) will be a newly allocated list of any arguments unresolved after all other actual arguments have been matched up against the formal arguments.

Parameters

Name	Type	Description
formals	symbols	The arguments associated with the specified procedure.
body	expressions	The list of expressions that define the behavior of the procedure.

Return Value

This expression returns the procedure to which the **lambda** expression evaluates.

Examples

```
(define                          ; define expression
    ave_3_nums                   ; symbol of define
    (lambda                      ; lambda procedure
        (x y z)                  ; lambda formals
        (/ (+ x y z) 3)          ; lambda body
    )                            ; end of lambda
)                                ; end of define
```

Executing this define causes the symbol **ave_3_nums** to be associated with a lambda expression, that is, a procedure. Once defined, **ave_3_nums** may be called like any other procedure. Given the definition above, the following expressions would evaluate as shown:

```
(ave_3_nums 2 5 8)         => 5
(ave_3_nums 3 6 (/ 18 2)) => 6
```

lambdaq

Syntax

```
(lambdaq formals body)
```

The formals can have one of the following three forms:

```
(variable1 ... )
variable
(variable1 ... variableN . variableN+1)
```

Description

lambdaq is identical to lambda (see **lambda** on page 315) except that it does not evaluate its arguments (*formals*) before executing the procedure.

Parameters

Name	Type	Description
formals	symbols	The arguments associated with the specified procedure.
body	expressions	The list of expressions that define the behavior of the procedure.

Return Value

This expression returns the procedure to which the **lambdaq** expression evaluates.

Examples

```
(define myfn
    (lambdaq (x y)
        (let
             ((a 10) (b 20) (c 30) (d 40))
             (+ (eval x) (eval y))
        )
    )
)
(myfn a b)      => 30
(myfn a c)      => 40
(myfn c d)      => 70
```

14.8 Comment

The comment functions is:

comment

Syntax

```
(comment title multi-linecomment)
```

Description

Documents Monk code. Has no runtime value.

Parameters

Name	Type	Description
title	string	A one-line description of the comment.
multi-linecomment	string	Complete description.

Return Value

None.

Example

```
(comment "Online Monitors"  "This section is optimized for the STC
Enterprise Montior. DO NOT CHANGE ANYTHING IN THIS SECTION!")
```

Chapter 15

Event Definitions

The Monk expressions listed below accept a structured event as a parameter. Each of these expressions is described in the following subsections.

$event-clear

Syntax

```
($event-clear event)
```

Description

Clears the data from the specified structured event.

Parameters

Name	Type	Description
event	structured event	Structured event to be cleared.

Return Value

Unspecified.

Examples

```
($event-clear output)          =>{MONK_UNSPECIFIED}

($event-clear input)           =>{MONK_UNSPECIFIED}
```

$event-parse

Syntax

```
($event-parse struct-definition string)
```

or

```
($event-parse struct-definition input-string-port)
```

Description

Maps event data into a structured event.

Parameters

Name	Type	Description
struct-definition	structured definition	The event type definition to map data into.
string	string	The data to map into your event.
input-string-port	port	Exact match on data only. Stops when no match occurs. Can be called until there is no data in *input-string-port*.

Return Value

Unspecified.

Examples

```
($event-parse input "data")        => {MONKUNSPECIFIED}

(define port (open-input-string "test"))
($event-parse input port)          => {MONKUNSPECIFIED}
```

$event->string

Syntax

```
($event->string event)
```

Description

Converts the data contained in a structured event into a string.

This function is usually located at the end of a collaboration function to generate a result (that is, the output event) to be returned by that collaboration function.

Use **$event->string** with a structured definition without data mapped to it for testing the structure.

Parameters

Name	Type	Description
event	structured event/structured definition	The variable name of the structured event or structured definition.

Return Value

string
A string representing the data contained in the structured event or structured definition.

Examples

In this example, the X_fix2dlm function creates an empty structured event *output* (using the **$make-event-map** expression), writes data to it (using the **copy-strip** expressions), then returns *output* as a string (using the **$event->string** expression).

```
;sample input "Simpson|Homer|Springfield|1980|10|31"
(load "fixedMsg.ssc")
(load "delimMsg.ssc")

(define X_fix2dlm
  (lambda (message-string)
    (let ((input
           ($make-event-map fixedMsg-delm fixedMsg-struct
                                       event-string))
          (output
           ($make-event-map delimMsg-delm
                                       delimMsg-struct)))
      (begin
       (copy-strip ~input%fixedMsg.LastName
                   ~output%delimMsg.CID.Name.LastName "")
       (copy-strip ~input%fixedMsg.FirstName
                   ~output%delimMsg.CID.Name.FirstName "")
       (copy-strip ~input%fixedMsg.Address
                   ~output%delimMsg.CID.Address "")
       (copy-strip ~input%fixedMsg.BirthYear
                   ~output%delimMsg.CID.Birthdate "")
       (copy-strip ~input%fixedMsg.BirthMonth
                   ~output%delimMsg.CID.Birthdate "")
       (copy-strip ~input%fixedMsg.BirthDay
                   ~output%delimMsg.CID.Birthdate "")
       )
      ($event->string output)  =>CID|Simpson^Homer|19801031)))
```

$make-event-map

Syntax

```
($make-event-map delim-list node_list [buffer])
```

Description

Creates an structured definition when the *buffer* is not specified. If the *buffer* is specified, a structured event is created.

Parameters

Name	Type	Description
delim-list	list	The list of delimiters that assist in parsing data into the structured event. See **Delimiter List** on page 41 for details.
node_list	event definition	Event definition description. See **Node List** on page 43 for details.
buffer	string	An optional data string that will be parsed into the structured event.

Return Value

A structured event or a structured definition.

Example for Identification Function

The **$make-event-map** expression is used in the variable bindings component of the following **let** expression. The **let** expression creates the environment for the **lambda** procedure. That environment is only accessible by elements of the **lambda**. An outline of a typical function used to identify an event by type is shown below.

```
(define IDfunction
(let ((input ($make-event-map delim-delm delim-struct)))
    (lambda (message-string)
      ($event-parse input message-string)
      (let ((result
             (and
             )))
        ($event-clear input)
        result
        ))))
```

When the identification function is called, the event is passed to the function and bound to the variable **message-string**.

The variable name “input” is later used in path expressions to reference locations within the event.

Example for Collaboration Function

An outline of a typical function used to collaborate an event follows.

```
(define Xlate-function
  (let ((input ($make-event-map delim-delm delim-struct))
        (output ($make-event-map delim-delm delim-struct))
       )
    (lambda (message-string)
```

```
($event-parse input message-string)
($event-clear output)
(begin
)
(let ((result ($event->string output)))
  ($event-clear input)
  ($event-clear output)
  result)
```

The variable name input is later used in path expressions to reference locations within the event.

The structured output event (bound to the variable output) initially has no content.

```
(define str "CID|Doe^Jane|123 Anywhere|19990101")
($make-event-map delimMsg-delm
                 delimMsg-struct str)    =>{MONK_ATOM_TYPE_EVENT}
```

$resolve-event-definition

Syntax

```
($resolve-event-definition node_list)
```

Description

Scans the *node_list* for templates, then replaces any template usage with the full event definition.

Parameters

Name	Type	Description
node_list	list	The quoted node list. See **Node List** on page 43.

Return Value

event
A resolved event definition.

Examples

```
;:- Global Template Reference
(load "CID.ssc")
;:- End Global Template Reference

;:- EvtStructure Definition
(define RAS-struct ($resolve-event-definition (quote
  (RAS ON 1 1 und und und und
    (MSH ON 1 1 "MSH" "MSH" und und)
    (NTE ON 0 INF "NTE" "NTE" und und)
    (CIDGRP OS 0 1 und und und und
      (CID GTN 1 1 "CID.ssc" CID-struct und und)
      (NTE ON 0 INF "NTE" "NTE" und und)
      (AL1 ON 0 INF "AL1" "AL1" und und)
      (PV1 ON 0 1 "PV1" "PV1" und und)
    )
  )
)))
;:- End Event Definition
```

The global (external file "CID.ssc") template is used to resolve the delimited node CID (GTN GLobal Template Node) and integrated with the RAS-struct node list.

change-pattern

Syntax

```
(change-pattern source-path destination change-list format)
```

Description

Copies the *source-path* path into the *dest* path while making substitutions according to *change-list*.

You can specify a series of input-pattern-to-output-string pairs (*change-list*), so that several conversions can take place in sequence. Optionally, you can use a regular expression to represent a pattern to match in the input.

Parameters

Name	Type	Description
source-path	string or path	The string or path to data in a structured event.
destination_path	path	The path to the data in the output event.
change-list	list	A list of the form: '((“input-pattern1” “output-string1”) (“input-pattern2” “output-string2”) … (“input-patternN” “output-stringN”))
format	string	An instruction to format the data for output. See **format** on page 107 for the syntax. Quotes are required, but can be empty (“ ”).

Return Value

Boolean

If any conversion took place, **#t**. If no conversion occurred, **#f** is returned.

Examples

```
;use change-pattern to expand an abbreviation
;sample input is LPC
;sample output is Laboratory Personnel Center
(display (change-pattern ~input%ORG.MSH.6  ~output%ORG.MSH.6'(("LPC"
"Laboratory Personnel Center"))""))

;exchange two characters, % for /
;at-sign (@) is a transitional, place-holding
;character, not found in source data
;sample input is %info%ab
;sample output is /info/ab
(change-pattern   ~input%fixed.ADT ~output%fixed.RX '(("%" "@")
("/" "%") ("@" "/"))"")

;remove leading zeros and trailing spaces
;sample inout is "0000123"
;sample ouput is "123"
(change-pattern ~input%strung.out ~output%trim.trunc '(("\^0\+" "")
("\+\$" ""))"%s")
```

```
;remove punctuation—parens, dash, x, X—from a phone
;number leaving only digits
;sample input is "(123)456-7890x1234"
;sample output is "12345678901234"
(change-pattern ~input%delim.0.3 ~output%fixed.1 '(("\[-()xX\]"
""))"")

;reformat name, delimited to fixed
;remove digits; exchange space for ^
;sample input is 5678^Manson^Louie^A
;sample output is Manson Louie A
(change-pattern ~input%ORD.0.3 ~output%DRO.3.1 '(("\[0-9\]" "" )
("^" ""))"")
```

Notes

If the data in the source matches *input-patternN*, then *output-stringN* is applied. If there are additional input-pattern/output-string pairs in the list, the output from the first is used as the input to the next, until all pairs have been processed in turn. The final result is written to the *destination_path*.

Because conversions are executed in the order listed, be sure to check input patterns carefully. If one input pattern matches part of another input pattern, place the longer pattern first. Otherwise, the longer pattern will never be matched (since the matching subpattern will already have been matched and replaced).

If the *source-path* data and the *input-pattern* don't match, no conversion takes place and an empty field or field element is written to a delimited output event. No data is written to a fixed event.

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

copy

Syntax

```
(copy source-path dest format)
```

Description

Copies data from the *source-path* to a *dest* path according to *format*.

Parameters

Name	Type	Description
source-path	string/path	The string or path to data in a structured event.
destination_path	path	The path into a structured event.
format	string	A control instruction to direct the format of the data for placement. See **format** on page 107 for the syntax. Quotes are required but can be empty (““).

Return Value

Unspecified.

Examples

```
;sample input is 'abc '
;sample output is 'abc '
(copy ~input%EVT.SE.0 ~output%ORG.CID.3 "") =>{MONK_UNSPECIFIED}
(copy ~input%EVT.NTE[0].3[1] ~output%RAS.OBXgrp[0].NTE[0].3[0] "%s"
)
```

Notes

The **copy** expression copies data as a string. It does not exchange delimiters within the string copied. If your destination delimiters differ the delimiters in the source-path data, use the **duplicate** expression.

This expression appends data if you do multiple copies to the same field without byte offset specified in the *destination_path*.

copy-strip

Syntax

```
(copy-strip source-path dest format)
```

Description

Copies data from the *source-path* to the *dest* path while removing ASCII-based trailing white space.

Parameters

Name	Type	Description
source-path	string/path	The string or path to data in a structured event.
destination_path	path	The path to the structured event.
format	string	A control instruction to direct the format of the data for placement. See **format** on page 107 for the syntax. Quotes are required but can be empty (“”).

Return Value

Unspecified.

Examples

```
;sample input is 'abc '
;sample output is 'abc '
(copy-strip SE.0  ~output%ORG.CID.3 "")  =>{MONK_UNSPECIFIED}
(copy-strip ~input%EVT.NTE[0].3[1  ~output%RAS.OBXgrp[0].NTE[0].3[0]
"%s")
```

Notes

The **copy-strip** expression copies data as a string. It does not exchange delimiters within the string copied. If your destination delimiters differ the delimiters in the source data, use the **duplicate** expression. This expression appends data if you do multiple copies to the same field without byte offset specified in the *destination_path*.

count-data-children

Syntax

```
(count-data-children path)
```

Description

Counts the number of child nodes that exist in the data tree of the structured event location specified by *path.*

Parameters

Name	Type	Description
path	path	The path to the structured event location to be counted.

Return Value

integer

The **count-data-children** expression returns the total number of instances (0 to *n*) of child nodes that are found in the data. If the child nodes are nonexistent, 0 is returned.

Examples

```
;Returns the actual number of SEG's children
(count-data-children ~input%EVT.SEG )

;SEG1 + SEG2 + SEG3 has three optional children nodes
(display ~input%EVT.SEG1)                          => aaa|bbb|ccc

(display (count-data-children ~input%EVT.SEG1)) => 3

(display ~input%EVT.SEG3)                          => a1|b2|c3|d4|e5|f6

(display (count-data-children ~input%EVT.SEG3)) => 6
```

count-map-children

Syntax

```
(count-map-children path)
```

Description

Counts counts the number of child nodes defined in the resolved event.

Note: *This function was formerly known as **count-children**.*

Parameters

Name	Type	Description
path	path	The path to the structured event location to be counted.

Return Value

integer

The **count-map-children** expression returns the total number of children (0 to *n*) defined in the resolved event.

Examples

```
;Returns the number of child nodes defined for SEG
(count-map-children ~input%EVT.SEG)

;SEG1, SEG2, SEG3 have three optional children nodes
(display ~input%EVT.SEG1)                          => aaa|bbb|ccc
(display (count-map-children ~input%EVT.SEG1))    => 3
(display ~input%EVT.SEG2)                          => 111|333
(display (count-map-children ~input%EVT.SEG2))    => 3
(display ~input%EVT.SEG3)                          => a1|b2|c3|d4|e5|f6
(display (count-map-children ~input%EVT.SEG3))    => 3
```

count-rep

Syntax

```
(count-rep path)
```

Description

Counts the total number of repetitions of the specified node that are found in the structured event data tree. Use this expression when writing expressions that loop on repeating event elements.

Parameters

Name	Type	Description
path	path	The path to the structured event element to be counted.

Return Value

integer
The **count-rep** expression returns the total number of repetitions (0 to *n)* of the specified node that are found in the event data tree.

Examples

```
;Returns the number of repetitions of the DTM segment
(count-rep ~input%EVT.DTM)

;Returns the number of repetitions of the REF segment
;in the third instance of the N1 group of event EVT
(count-rep  ~input%EVT.N1[2].REF)

(display ~input%EVT.DTM) =>DTM/one^MDTM/two^M
(display (count-rep ~input%EVT.DTM))  => 2
(display ~input%EVT.NT1[2])   => N1|AAA|REM^one|REM^two|REM^three|CCC
(display (count-rep ~input%EVT.N1C2].REM))  => 3
```

The most frequent application of the **count-rep** expression is in the **do** expression where it sets the maximum value for iterations of the loop and is compared to the iteration count in the **do** expression *test*. This is shown in the sample below.

```
(do ((i 0 (+ i 1))) ((>= i (count-rep ~input%Msg-In.PL)))
    (copy-strip ~input%Msg-In.NAM
             ~output%Msg-Out.Detail-Set[<i>].NAM "")
    (copy-strip ~input%Msg-In.PL[<i>]
             ~output%Msg-Out.Detail-Set[<i>].PL "")
)
```

data-map

Syntax

```
(data-map source-path destination_path filename format trim-chars)
```

Description

Matches a string to a string stored in an ASCII text file. The data associated with the matching string is inserted into the structured event.

Parameters

Name	Type	Description
source-path	string/path	The string or path to the data in a structured event.
destination_path	path	The path to the data in the structured event.
filename	string	The name of the file containing the matching data, including its absolute directory location. For example: **/home/user1/data/data-map.** The data file is an ASCII text file containing one *matchstring* and *mapped-data* pair per line, as discussed below.
format	string	An instruction to format the data before placement. See **format** on page 107 for the syntax. Quotes are required, but can be empty (“ ”).
trim-chars	string	Any leading or trailing characters to be trimmed from the *source* data before matching against a *matchstring*. All *trim-chars* are interpreted as literals. Quotes are required, but can be empty (“ ”).

Data File Specifications

Entries in the **data-map** data file have the format:

```
matchstring, mapped-data
```

For example:

```
Dr. John Edwards,(818)555-1564
Dr. Jane Docen,(302)555-1823
```

If no match to the source data is found in the data file, a default value entry is written to the output event. The syntax for the default value entry is shown below. Both lines are equivalent.

```
%default%,mapped-data
,mapped-data
```

where *mapped-data* is the data to be output. For example:

```
%default%,NO-MATCH
```

Because a comma is used as the delimiter in the data file, a comma must be preceded by a backslash (\,) if it appears in either the *matchstring* or *mapped-data*.

To represent a backslash in the data, enter two backslashes (\\).

A backslash before a NewLine character at the end of a data file line is interpreted as a literal and the NewLine character is written to the output event.

Return Value

Unspecified.

Examples

```
;;;the format quotes are empty
;;;the trim-chars quotes contain a space char
(datamap ~input%EVT.SE.0 ~output%ORG.CID.3
         "/home/user1/data/datamap.dat" "" " ")
  => {MONK_UNSPECIFIED}
```

Notes

The data in the *source-path* is matched against each *matchstring* in the *filename* data file. If a match is found, then the associated *mapped-data* is written to the *destination_path*.

If no match is found and there is a default value entry in the data file, the *mapped-data* for the default entry is written to the *destination_path*.

If no match is found and there is no default value entry in the data file, an exception is returned and the **data-map** function fails.

If the string in the event may be padded with leading or trailing spaces, use the *trim-chars* parameter to ensure that the *matchstring* matches the *source-path* data.

display-event-data

Syntax

```
(display-event-data event [port])
```

Description

Displays the data in the specified Event. For each node in the Event, the node's data and information about that data is displayed on a single line using the following format:

```
(Depth:Length:Children:FLAGS)      :Data
```

The indentation shows the level at which the data resides in the Event structure—more indented means further down in the structure.

Table 2 Key to Data Line Values

Name	Description
Depth	The level in the Event structure where the data resides.
Length	The number of bytes of data.
Children	The number of child nodes associated with this node.
FLAGS	Any of the following: R—Repetition node D—Data A—Arrayified (the data is internally compressed) C—Constant B—ChildData (the child nodes have data) S—SibData (the sibling nodes have data)
Data	The actual data.

Parameters

Name	Type	Description
event	event_struct	The Event to be displayed.
port	port	Optional. The port to which the Event data is displayed. If no port is specified, the Event data is sent to standard out. **Note:** the use of display formatting characters, such as the carriage return character "\r", in the data will affect how the data is displayed when it is sent to the screen.

Return Value

Unspecified.

Example

```
(define MonkExample-delm '(
        ("*" endofrec)
        ("|")
        ("~" array)
        ("^")
        ("&")))

(define MonkExample-struct ($resolve-event-definition (quote
            (MonkExample ON 1 1 und und und -1
                (Name ON 1 1 und und und -1)        ;:= {0.0:N}
                (Address ON 1 1 und und und -1)     ;:= {0.1:N}
            )                                       ;:= {0:N}
)))

(define MonkExample-data "Ese Bodyne*404 Huntington Dr.*")

(define MonkExOut (open-output-file "MonkExampleOutput.dat"))

(define MonkExample-event ($make-event-map MonkExample-delm
        MonkExample-struct))

($event-parse MonkExample-event MonkExample-data)

(display-event-data MonkExample-event MonkExOut)

=>Unspecified
```

The file **MonkExampleOutput.txt** now contains:

```
(Depth:Length:Children:FLAGS(Rep,Data,Arrayified,Constant,ChildData,
SibData))
(0:30:1:DACB)    :Ese Bodyne*404 Huntington Dr.*
(1:30:2:RDACB)    :Ese Bodyne*404 Huntington Dr.*
(2:10:1:DACB)      :Ese Bodyne
(3:10:0:RDAC)       :Ese Bodyne
(2:18:1:DACB)      :404 Huntington Dr.
(3:18:0:RDAC)       :404 Huntington Dr.
```

display-event-dump

Syntax

```
(display-event-dump event [port])
```

Description

This function combines the two functions **display-event-data** and **display-event-map**. It displays the data in the specified Event along with the Event structure. For each node in the Event, information about the node's structure is displayed first on a single line, then the data in the node and information about that data is displayed on the next line using the following format:

```
((Modifiers):Name:Type:MinRep:MaxRep:Tag:Def:Offset:(Length|Encoding)
:Delim:BitFlags)
```

```
(Depth:Length:Children:FLAGS)      :Data
```

The indentation shows the level at which the data resides in the Event structure—more indented means further down in the structure.

Important: *The following table briefly identifies the type of structure information displayed. For a complete discussion of the various values returned see* **"Node List" on page 43**.

Table 3 Key to Structure Line Values

Name	Description
Modifiers	Any of the following: Bd—Begin delimiter Ed—End delimiter Ri—Array repetition information Ex—Exact map (not extended) Gr—Group child repetitions Co—Consumer Get—Get function NofN—Min/Max children Put—Put function Sc—Scavenger string ScN—Scavenger string with no first character Nt—Not tagged (data doesn't match tag character)
Name	The name of the node.

Table 3 Key to Structure Line Values

Name	Description
Type	The type of node. Any of the following: ON—Delimited AN—Any-ordered delimited node OF—Fixed node AF—Any-ordered fixed node OS—Ordered set AS—Any-ordered set ONA—Ordered delimited node-array ANA—Any-ordered delimited node-array GTN—Global template, delimited node LTN—Local template, delimited node GTF—Global template, fixed node LTF—Local template, fixed node GTS—Global template, set LTS—Local template, set
MinRep	The minimum number of repetitions of the node.
MaxRep	The maximum number of repetitions of the node.
Tag	Tag character.
Def	Default data.
Offset	Byte offset.
Length|Encoding	Length or encoding.
Delim	Delimiter.
BitFlags	Any of the following: Su—Strongly unique Wu—Weakly unique Nu—Not unique RNu—Required, not unique Dc—Defined children Pd—Parent delimited Lr—Length rest Loc—Local delimiters Dd—Default path Le—Length encoded Bdm—Beyond defined map Ao—Any ordered

Table 4 Key to Data Line Values

Name	Description
Depth	The level in the Event structure where the data resides.
Length	The number of bytes of data.
Children	The number of child nodes associated with this node.
FLAGS	Any of the following: R—Repetition node D—Data A—Arrayified (the data is internally compressed) C—Constant B—ChildData (the child nodes have data) S—SibData (the sibling nodes have data)
Data	The actual data.

Parameters

Name	Type	Description
event	event_struct	The Event to be displayed.
port	port	Optional. The port to which the Event data is displayed. If no port is specified, the Event data is sent to standard out. **Note:** the use of display formatting characters, such as the carriage return character “\r”, in the data will affect how the data is displayed when it is sent to the screen.

Return Value

Unspecified.

Example

```
(define MonkExample-delm '(
        ("*" endofrec)
        ("|")
        ("~" array)
        ("^")
        ("&")))

(define MonkExample-struct ($resolve-event-definition (quote
            (MonkExample ON 1 1 und und und -1
                (Name ON 1 1 und und und -1)        ;:= {0.0:N}
                (Address ON 1 1 und und und -1)     ;:= {0.1:N}
            )                                       ;:= {0:N}
)))

(define MonkExample-data "Ese Bodyne*404 Huntington Dr.*")

(define MonkExOut (open-output-file "MonkExampleOutput.dat"))

(define MonkExample-event ($make-event-map MonkExample-delm
        MonkExample-struct))

($event-parse MonkExample-event MonkExample-data)

(display-event-dump MonkExample-event MonkExOut)

=>Unspecified

The file MonkExampleOutput.txt now contains:

((Modifiers):Name:Type:MinRep:MaxRep:Tag:Def:Offset:(Length|Encoding)
:Delim:BitFlags)
               (():MonkExample:ON:1:1:::-1:(-1)::Su,Dc)
(Depth:Length:Children:FLAGS(Rep,Data,Arrayified,Constant,ChildData,
SibData))
(0:30:1:DACB)     :Ese Bodyne*404 Huntington Dr.*
                 (():MonkExample:ON:1:1:::-1:(-1)::Su,Dc)
(1:30:2:RDACB)     :Ese Bodyne*404 Huntington Dr.*
                  (():Name:ON:1:1:::-1:(-1):"*":RNu)
(2:10:1:DACB)       :Ese Bodyne
                   (():Name:ON:1:1:::-1:(-1):"*":RNu)
(3:10:0:RDAC)        :Ese Bodyne
                  (():Address:ON:1:1:::-1:(-1):"*":RNu)
(2:18:1:DACB)       :404 Huntington Dr.
                   (():Address:ON:1:1:::-1:(-1):"*":RNu)
(3:18:0:RDAC)        :404 Huntington Dr.
```

display-event-map

Syntax

```
(display-event-map event [port])
```

Description

This function displays the structure for the specified Event. For each node in the Event, information about the node's structure is displayed on a single line using the following format:

```
((Modifiers):Name:Type:MinRep:MaxRep:Tag:Def:Offset:(Length|Encoding)
:Delim:BitFlags)
```

The indentation shows the level at which the node resides in the Event structure—more indented means further down in the structure.

Important: *The following table briefly identifies the type of structure information displayed. For a complete discussion of the various values returned see* **"Node List" on page 43**.

Table 5 Key to Structure Line Values

Name	Description
Modifiers	Any of the following: Bd—Begin delimiter Ed—End delimiter Ri—Array repetition information Ex—Exact map (not extended) Gr—Group child repetitions Co—Consumer Get—Get function NofN—Min/Max children Put—Put function Sc—Scavenger string ScN—Scavenger string with no first character Nt—Not tagged (data doesn't match tag character)
Name	The name of the node.
Type	The type of node. Any of the following: ON—Delimited AN—Any-ordered delimited node OF—Fixed node AF—Any-ordered fixed node OS—Ordered set AS—Any-ordered set ONA—Ordered delimited node-array ANA—Any-ordered delimited node-array GTN—Global template, delimited node LTN—Local template, delimited node GTF—Global template, fixed node LTF—Local template, fixed node GTS—Global template, set LTS—Local template, set

Table 5 Key to Structure Line Values

Name	Description
MinRep	The minimum number of repetitions of the node.
MaxRep	The maximum number of repetitions of the node.
Tag	Tag character.
Def	Default data.
Offset	Byte offset.
Length|Encoding	Length or encoding.
Delim	Delimiter
BitFlags	Any of the following: Su—Strongly unique Wu—Weakly unique Nu—Not unique RNu—Required, not unique Dc—Defined children Pd—Parent delimited Lr—Length rest Loc—Local delimiters Dd—Default path Le—Length encoded Bdm—Beyond defined map Ao—Any ordered

Parameters

Name	Type	Description
event	event_struct	The Event to be displayed.
port	port	Optional. The port to which the Event data is displayed. If no port is specified, the Event data is sent to standard out. **Note:** the use of display formatting characters, such as the carriage return character “\r”, in the data will affect how the data is displayed when it is sent to the screen.

Return Value

Unspecified.

Example

```
(define MonkExample-delm '(
        ("*" endofrec)
        ("|")
        ("~" array)
        ("^")
        ("&")))

(define MonkExample-struct ($resolve-event-definition (quote
            (MonkExample ON 1 1 und und und -1
                (Name ON 1 1 und und und -1)         ;:= {0.0:N}
                (Address ON 1 1 und und -1)          ;:= {0.1:N}
            )                                        ;:= {0:N}
)))

(define MonkExample-data "Ese Bodyne*404 Huntington Dr.*")

(define MonkExOut (open-output-file "MonkExampleOutput.dat"))

(define MonkExample-event ($make-event-map MonkExample-delm
        MonkExample-struct))

($event-parse MonkExample-event MonkExample-data)

(display-event-map MonkExample-event MonkExOut)

=>Unspecified
```

The file **MonkExampleOutput.txt** now contains:

```
((Modifiers):Name:Type:MinRep:MaxRep:Tag:Def:Offset:(Length|Encoding)
:Delim:BitFlags)

(():MonkExample:ON:1:1:::-1:(-1)::Su,Dc
 (():Name:ON:1:1:::-1:(-1):"*":RNu
  (():undefined:ON:1:1:::0:(0):"|":Bdm,Nu
   (():undefined:ON:1:1:::0:(0):"^":Bdm,Nu
    (():undefined:ON:1:1:::0:(0):"&":Bdm,Nu))))
 (():Address:ON:1:1:::-1:(-1):"*":RNu
  (():undefined:ON:1:1:::0:(0):"|":Bdm,Nu
   (():undefined:ON:1:1:::0:(0):"^":Bdm,Nu
    (():undefined:ON:1:1:::0:(0):"&":Bdm,Nu))))
)
```

duplicate

Syntax

```
(duplicate source-path destination_path format)
```

Description

Copies leaf data from the *source-path* to the corresponding leaf positions of the destination structured event. Leaf data is contained in nodes without children. This function overwrites any existing data in the location.

Parameters

Name	Type	Description
source-path	path	The path to the data in a structured event.
destination_path	path	The path to the data in the structured event.
format	string	An instruction to format the data for output. See **format** on page 107 for the syntax. Quotes are required, but can be empty (“ ”).

Return Value

Unspecified.

Examples

```
;sample input is 'abc '
;sample output is 'abc '
(duplicate ~input%EVT.SE.0 ~output%ORG.CID.3 "")
(duplicate ~input%EVT.NTE[0].3[1] ~output%RAS.OBXgrp[0].NTE[0].3[0]
"%s")
                        => {MONK_UNSPECIFIED}
```

duplicate-strip

Syntax

```
(duplicate-strip source-path destination_path format)
```

Description

Copies leaf data from the *source-path* to the corresponding leaf positions of the destination structured event, after removing trailing spaces for data at the leaf to be duplicated. Leaf data is contained in nodes without children. This function overwrites any existing data in the leaf locations.

Parameters

Name	Type	Description
source-path	path	The path to the data in a structured event.
destination_path	path	The path to the data in the structured event.
format	string	An instruction to format the data for output. See **format** on page 107 for the syntax. Quotes are required, but can be empty (" ").

Return Value

Unspecified.

Examples

```
;sample input is 'abc '
;sample output is 'abc '
(duplicate-strip ~input%EVT.SE.0 ~output%ORG.CID.3 " ")
(duplicate-strip ~input%EVT.NTE[0].3[1]
~output%RAS.OBXgrp[0].NTE[0].3[0] "%s")
                        => {MONK_UNSPECIFIED}
```

file-check

Syntax

```
(file-check source filename)
```

Description

Compares the file contents against the source data.

Parameters

Name	Type	Description
source	string/path	The data to be compared.
filename	string	The name of the file to compare, including its absolute directory location, for example, **/home/user1/filename**

Return Value

Boolean

This expression returns **#t** if the files are equal. Otherwise, it returns **#f**.

Examples

```
;Compares the contents of the SEG node with the
;contents of the specified file
(file-check ~input%EVT.SEG "/home/user1/filename")

;contents of filename: hello
(file-check "hello" "/home/user1/filename")      => #t
(file-check "bye" "/home/user1/filename")        => #f
```

file-lookup

Syntax

```
(file-lookup source filename)
```

Description

Matches source data against data contained in a filename. The data in the *source* location is compared to the strings in *filename*.

Parameters

Name	Type	Description
source	path/string	The data to compare.
filename	string	The name of the file containing the matching data, including its absolute directory location. For example: **/home/user1/data/dept_phone.** The data file is an ASCII text file containing one string per line. For example: **10099** **10100** **10104** **10211** **11307** The maximum string length is 8,096 bytes.

Return Value

If a match is found, the expression returns **#t**. If no match is found, the expression generates an exception and the function fails.

Examples

```
(file-lookup ~input%EVT.EVN.1 "/home/user1/data/events")    => #t
```

get

Syntax

```
(get path)
```

Description

Extracts data from a structured event.

Parameters

Name	Type	Description
path	path	The path to the data in the structured event.

Return Value

string

The **get** expression returns a string representing the data in the *path* location.

Examples

```
;get Event Type Code and compare to string "A01"
(regex "A01" (get ~input%ORG.EVN.ETC))
;get Current Balance, convert to a number, and
;check that it's greater than 0
(>(string->number(get ~input%ORG.PV1.46)) 0)

;sample input is hello
(get ~input%ORG.EVN.ETC)                                => hello
```

list-lookup

Syntax

```
(list-lookup source-path destination_path lookup-list format
trim-chars)
```

Description

Matches data in the source-path against the key elements of a list and copies the associated value into the structured event.

Parameters

Name	Type	Description
source-path	path	The data to match.
destination_path	path	The path to the structured event destination.
lookup-list	list	A list of the form: '((“matchstring1” “output-string1”) (“matchstring2” “output-string2”) ... (“matchstringN” “output-stringN”) (else “default-output-string”) Can also be a variable name that has a value of a list of this form. See Notes below.
format	string	An instruction to format the data for output. See **format** on page 107 for the syntax. Quotes are required, but can be empty (“”).
trim-chars	string	Any leading or trailing characters to be trimmed from the *source* data before matching against the *lookup-list*. All *trim-chars* are interpreted as literals. Quotes are required, but can be empty (“”).

Return Value

Unspecified.

Examples

```
;;;the trim-chars quotes contain a space char
;sample input is ADD
;sample output is A01
(list-lookup ~input%EVT.SE.0 ~output%ORG.CID.3 '(("ADD" "A01")
("MOVE" "A02")("DELETE" "A03")(else "DONTKNOW")) "%s" "")
                            => {MONK_UNSPECIFIED}
```

Notes

The data in the *source-path* is matched against each *matchstring*. If a match is found, then the associated *output-string* is written to the *destination_path*. If no match is found, then the *default-output-string* is written to the *destination_path*. If no match is found and there is no *default-output-string*, it will error out.

path-node-has-data?

Syntax

```
(path-node-has-data? path)
```

Description

Verifies whether or not the specified path location of a structured event contains data.

Parameters

Name	Type	Description
path	path	The path to the event element to be verified.

Return Value

Boolean

This expression returns **#t** if the specified path location contains data. Otherwise, it returns **#f**.

Examples

```
;Verifies whether the SEG node contains data
;sample input is "aaa|bbb"
(path-node-has-data? ~input%EVT.SEG)                => #t

(path-node-has-data? ~input%EVT.SEG.three)          => #f

;sample input is "111||333"
;optional node with tag
(path-node-has-data? ~input%EVT.SEG.two)            => #f

;sample input is "111||333"
;optional set
(path-node-has-data? ~input%EVT.SEG.two)            => #f

;sample input is "111||333"
;optional node without tag
(path-node-has-data? ~input%EVT.SEG.two)             => #t
```

Notes

If an optional node has no tag and the input data for that node ends with a delimiter, this function will return **#t** since the empty string is valid.

not-verify

Syntax

```
(not-verify path reg_exp)
```

Description

Matches data against a regular expression. The **not-verify** expression is the complement of verify.

Parameters

Name	Type	Description
path	path	The path to the data to be verified.
reg_exp	string	A regular expression. See **"Regular Expressions" on page 29** for the regular expression syntax.

Return Value

Boolean

If no *exact* match is found, **#t** is returned. If an *exact* match is found, an exception is generated.

Examples

```
;check a location for an empty field
;("\.\+" matches any string of at least one character)
;sample input is "Hello"
(not-verify ~input%EVT.SE.0 "\.\+")        => error

;check a location for a specific string
(not-verify ~input%RAS.CID.8 "F")          => #t

;match location's contents against a regular expression
;this expression checks for a Social Security Number
;sample input is "(111) 222-3333"
(not-verify ~input%RAS.CID.19
      "\[0-9\]\{3\}-\[0-9\]\{2\}-\[0-9\]\{4\}")
          => #t

;match location's contents against a regular expression
;this expression checks for one of a set of strings
;sample input is "CA"
(not-verify~input%RAS.CID.11[0].3 "CA\|OR\|WA"
          => error
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

path?

Syntax

```
(path? object)
```

Description

Tests the object to determine whether or not it represents a path.

Parameters

Name	Type	Description
object	any	The object to test.

Return Value

Boolean

This expression returns **#t** if object represents a path. Otherwise, it returns **#f**.

Examples

```
;The following example returns #f
(path? "EVT.SEG")                      => #f
;
;The following example returns #t
(path?  ~input%EVT.SEG)                => #t
```

path-defined?

Syntax

```
(path-defined? object)
```

Description

Tests the object to determine whether or not it exists as a path in an ETD.

Parameters

Name	Type	Description
object	any	The object to test.

Return Value

Boolean

This expression returns **#t** if object represents a valid ETD path. Otherwise, it returns **#f**.

Examples

```
(define path_str "~input%root.unknownnode")
(path-defined? (string->path path_str))     =>     #f
(path-defined? ~input%root.firstnode)       =>     #t
```

path-defined-as-repeating?

Syntax

```
(path-defined-as-repeating? path)
```

Description

Verifies whether the specified node is defined as repeating in the event definition.

Parameters

Name	Type	Description
path	path	The structured event element to be tested.

Return Value

Boolean

This expression returns **#t** if the specified node is defined as repeating. Otherwise, it returns **#f**.

Examples

```
;The following will return #f because it is verifying a
;root node which cannot be defined as repeating.
;
(path-defined-as-repeating? ~input%EVT)    => #f
```

path-event

Syntax

```
(path-event path)
```

Description

Gets the Event associated with the specified path.

Parameters

Name	Type	Description
path	path	A complete path.

Return Value

event_struct
Returns the Event associated with the specified path.

Example

```
(define MonkExample-delm '(
        ("*" endofrec)
        ("|")
        ("~" array)
        ("^")
        ("&")))

(define MonkExample-struct ($resolve-event-definition (quote
            (MonkExample ON 1 1 und und und -1
                (Name ON 1 1 und und und -1)        ;:= {0.0:N}
                (Address ON 1 1 und und und -1)     ;:= {0.1:N}
            )                                       ;:= {0:N}
)))

(define MonkExample-data "Ese Bodyne*404 Huntington Dr.*")

(define MonkExample-event ($make-event-map MonkExample-delm
        MonkExample-struct))

($event-parse MonkExample-event MonkExample-data)

(display (path-event ~MonkExample-event%MonkExample.Name))

=>{MONK_ATOM_TYPE_EVENT}
```

path-event-symbol

Syntax

```
(path-event-symbol path)
```

Description

Gets the symbol that represents the Event structure for the specified path.

Parameters

Name	Type	Description
path	path	A complete path.

Return Value

symbol
The symbol representing the Event associated with the specified path.

Example

```
(define MonkExample-delm '(
        ("*" endofrec)
        ("|")
        ("~" array)
        ("^")
        ("&")))

(define MonkExample-struct ($resolve-event-definition (quote
            (MonkExample ON 1 1 und und und -1
                (Name ON 1 1 und und und -1)       ;:= {0.0:N}
                (Address ON 1 1 und und und -1)    ;:= {0.1:N}
            )                                      ;:= {0:N}
)))

(define MonkExample-data "Ese Bodyne*404 Huntington Dr.*")

(define MonkExample-event ($make-event-map MonkExample-delm
        MonkExample-struct))

($event-parse MonkExample-event MonkExample-data)

(display (path-event-symbol ~MonkExample-event%MonkExample.Name))

=>MonkExample-event
```

path-nodeclear

Syntax

```
(path-nodeclear path)
```

Description

Deletes all the data from the specified node and marks the node as containing no data.

Parameters

Name	Type	Description
path	path	The path of the node to clear.

Return Value

Unspecified.

Examples

```
(path-nodeclear ~input%root.an.friend)
```

path-nodedepth

Syntax

```
(path-nodedepth path)
```

Description

Determines the depth of the node indicated by the path parameter. The depth is calculated from the root node.

Parameters

Name	Type	Description
path	path	The structured event element to be tested.

Return Value

integer
This expression returns an integer of 0 or more.

Examples

```
;The following example would return a result of 3.

(path-nodedepth ~input%EVT.SEG.A)    => 3

(path-nodedepth ~input%EVT)          =>1

(path-nodedepth ~input)              =>0
```

path-nodename

Syntax

```
(path-nodename path [depth])
```

Description

Provides the name of the node in the event definition indicated by the path parameter.

Parameters

Name	Type	Description
path	path	The structured event element to be tested.
depth	integer	Optional parameter giving the depth for the name.

Return Value

symbol

This expression returns the node name for the indicated path. If the depth is not specified, this expression returns the last element.

Examples

```
;The following example returns "EVT"
(path-nodename ~input%EVT)
;
;The following example returns "SEG"
(path-nodename ~output%EVT.SEG.field 2)
```

path-nodeparentname

Syntax

```
(path-nodeparentname path grandparent)
```

Description

Provides the parent node name from the specified path and depth.

Parameters

Name	Type	Description
path	path	The structured event element to access.
grandparent	integer	Optional parameter specifying the number of levels above the last node in the path.

Return Value

symbol

This expression returns the parent node name from the specified path. If no integer is specified, this expression returns the parent of the child. If an integer is specified, this expression returns the parent node name at the number of nodes above the child. If the integer specified is greater than the depth of the path, **#f** is returned.

Examples

```
(path-nodeparentname ~input%EVT.SEG1)        => EVT

(path-nodeparentname ~output%EVT.SEG1.SEG2.SEG3 6)   => #f
```

path-put

Syntax

```
(path-put source destination [format])
```

Description

Similar to **copy**, in that it places the source data into the Event at the location specified in the destination. The important difference is that copy only works with strings, but path-put works with other Monk data types.

Important: *If the source is not a "string" then the node specified in the destination path must have the **put** modifier set in order for this function to complete successfully. The **put** node modifier converts the source argument to a string before placing the data into the Event. See* **"Node List" on page 43** *for more information on node modifiers.*

Parameters

Name	Type	Description
source	any	The data you want to place in the destination node. **Note:** If the source is not a "string" then the node specified in the destination path must have the "put" modifier set. See **"Node List" on page 43** for more information on node modifiers.
destination	path	A complete path.
format	string	Optional. A valid format specification. See **"Format Specification" on page 34** for information on formatting output.

Return Value

Unspecified.

path->string

Syntax

```
(path->string path)
```

Description

Converts the specified path to a string.

Parameters

Name	Type	Description
path	path	The path to convert to a string.

Return Value

string
The string conversion of the path.

Examples

```
(path->string ~input%MSG)                  =>     "~input%MSG"

(string? (path->string ~input%MSG))        =>     #t
```

path-valid?

Syntax

```
(path-valid? path)
```

Description

Verifies that the path specified is valid for the structured event.

Parameters

Name	Type	Description
path	path	The path to be verified.

Return Value

Boolean

This expression returns **#t** if the specified path is valid in the event type definition. Otherwise, it returns **#f**.

Example

```
(path-valid? ~input%EVT)          => #t
```

This function call will evaluate as shown if the path EVT exists in the input structure.

string->path

Syntax

```
(string->path string)
```

Description

Converts the contents of the specified string to a path or partial path.

The path is unresolved. To resolve the path in the desired environment, you may need to perform an **eval**.

Parameters

Name	Type	Description
string	string	The characters to convert to a path.

Return Value

path
Newly-created unresolved path.

Examples

```
(string->path "~input%MSG")            =>      ~input%MSG

(path? (string->path "~input%MSG")    =>      #t
```

timestamp

Syntax

```
(timestamp destination_path timeformat)
```

Description

Inserts the current date and time (of the server's host system) into the structured event. You can specify a custom format or use the default format.

If you give **timestamp** an empty string, it will output nothing.

Parameters

Name	Type	Description
destination_path	path	The path to the structured event element for placement.
timeformat	string	An instruction to format the data. Syntax is detailed below. Quotes are required, but can be empty (" ").

The **timeformat** can include one or more of the following format choices. Text can be included, for example, "time test-%r" generates the output, "time test-02:15:03 PM".

Time Division	Format Option	Description	Value Range or Sample Output
Days	**%w**	day of week (Sunday is day 0)	0–6
	%a	day of week, using site-defined abbreviations	for example, Sun, Mon, Tue, and so forth.
	%A	day of week, using site-defined spellings	for example, Sunday, Monday, and so forth.
	%d	day of month	01–31
	%e	day of month (single digits are preceded by a space)	1–31
	%j	day of year	001–366
Weeks	**%U**	week of year (Sunday is the first day of the week)	01–52
	%W	week of year (Monday is the first day of the week)	01–52
Months	**%m**	month number	01–12
	%b	month, using site-defined abbreviations	for example, Jan, Feb, Mar, Apr, and so forth.
	%B	month, using site-defined spellings	for example, January, February, and so forth.
Years	**%y**	year within century	00–99
	%Y	year, including century	for example, 1988

Time Division	Format Option	Description	Value Range or Sample Output
Hours	**%H**	hour	00–23
	%I	hour	00–12
	%k	hour (single digits are preceded by a space)	0–23
	%l	hour (single digits are preceded by a space)	1–12
Minutes	**%M**	minute	00–59
Seconds	**%S**	seconds	00–59
Morning or Afternoon	**%p**	AM or PM	AM or PM
Time Zone	**%Z**	time zone abbreviation	for example, PDT
Composites	**%D**	date as %m/%d/%y	for example, 02/05/04
	%R	time as %H:%M	for example, 14:15
	%T	time as %H:%M:%S	for example, 14:15:03
	%r	time as %I:%M:%S %p	for example, 02:15:03 PM
	%x	site-defined standard date format	for example, 09/12/93

Example

```
;Current date/time is March 5, 1995, 4:15 p.m.
;sample output is ""
(timestamp ~output%ORG.CID.3  "" ) => {MONK_UNSPECIFIED}

;Current date and time is March 5, 1995, 4:15:03 p.m.
;sample output is current time:03/05/95, 04.15.03PM
(timestamp ~output%ORG.CID.3
           "current time:%D, %H.%M.%S%p" )
                    => current time:03/05/95, 04.15.03PM
```

uniqueid

Syntax

```
(uniqueid path)
```

Description

Creates a unique identifier string. The identifier string is based upon the current system time, day, month, and year to a string.

Parameters

Name	Type	Description
path	path	Where to write the string in the output event.

Return Value

Unspecified.

Examples

```
;The uniqueid data is written to the SEG node
(uniqueid ~output%EVT.SEG )      => {MONK_UNSPECIFIED}

(display ~output%EVT.SEG )       => 2000012714152908 54
```

Note: *Although the uniqueid function provides a properly unique identifier, it should not be used as a time-stamp. For time-stamp functionality, see* **timestamp** *on page 365.*

verify

Syntax

```
(verify path reg_exp)
```

Description

Matches data against a regular expression.

A regular expression can be used to:

- check if a field is empty
- match a specified string
- match from a set of strings.

Parameters

Name	Type	Description
path	path	The path to data to be verified.
reg_exp	expression	A regular expression. See **"Regular Expressions" on page 29** for the regular expression syntax.

Return Value

If an *exact* match is found, **#t** is returned. If an *exact* match is not found, an exception is generated.

Examples

```
;check a location for a non-empty field
;(".\+" matches any string of at least one character)
;sample input is Hello
(verify ~input%EVT.SE.0 ".\+" )      => #t

;check a location for a specific string
(verify ~input%RAS.CID.8 "F")        => #f

;match a location's contents against a regular expression
;this expression checks for a SSN
;sample input is 111-22-3333
(verify ~input%RAS.CID.19 "\[0-9\]\{3\}-\[0-9\]\{2\}-\[0-9\]\{4\}"
)                                    => #t

;match a location's contents against a regular expression
;this expression checks for one of a set of strings
;sample input is "CA"
(verify ~input%RAS.CID.11[0].3 "CA\|OR\|WA"
)                                    => #t
```

Note: *The return values may vary on different platforms due to the differences between ASCII and EBCDIC values.*

Chapter 16

Date and Time

The Date and Time functions include:

difftime

Syntax

```
(difftime time1 time2)
```

Description

Calculates the difference between two time arguments.

Parameters

Name	Type	Description
time1 time2	integer	Use the parameters specified in **mktime** on page 373 or **time** on page 376 function to set the time parameters. See the description of these functions in this section for further details.

Return Value

integer
Number of seconds difference between *time1* and *time2*.

Examples

```
(difftime (mktime 99 01 01 12 30 43 1)
          (mktime 99 01 01 12 30 33 1)) => 10
(difftime (time)
          (mktime 99 01 01 12 30 33))   => 6225903
(difftime (mktime 99 01 01 12 30 33)
          (time))                       => 6225903
(difftime (mktime 99 01 01 12 30 33)
          (mktime 100 01 01 12 30 33))  => -31536000
```

gregorian_date->julian_days

Syntax

```
(gregorian_date->julian_days date)
```

Description

Converts a gregorian date to a julian days.

Parameters

Name	Type	Description
date	string	Integers in the format **YYYYMMDD**, where: **YYYY** is the year. **MM** is the month **DD** is the day

Return Value

integer
This function returns the julian days calculated. If no conversion occurs, **#f** is returned.

Examples

```
(gregorian_date->julian_days "-47131124")  => 0

(gregorian_date->julian_days "20000101")   => 2451545

(gregorian_date->julian_days "99350")      => #f
```

julian_days->gregorian_date

Syntax

```
(julian_days->gregorian_date days)
```

Description

Converts julian days to a gregorian date (YYYYMMDD).

Parameters

Name	Type	Description
days	integer, string, or number	A valid julian date.

Return Value

number

This function returns the gregorian date calculated, or **#f** if no conversion possible.

Examples

```
(julian_days->gregorian_date "0")       => -47131124

(julian_days->gregorian_date 2451545)   => 20000101
```

mktime

Syntax

```
(mktime year month day hour minute seconds [DST])
```

Description

Creates a Monk-time object from the specified parameters.

Parameters

Name	Type	Description
year	integer	The year minus 1900. 69-138 years. Must be between 69 (representing the year 1969) and 138 (representing the year 2038).
month	integer	The numeric month minus one. (0=Jan, 1=Feb, ... 11=Dec).
day	integer	The day of the month. 0-31 days. 0 = the last day of the previous month.
hour	integer	The hour of the day in 24 hour time. 0-23 hours.
minute	integer	The minute of the hour. 0-59 minutes.
seconds	integer	The second of the minute. 0-59 seconds.
DST	integer	Optional. Compensates for daylight savings time (DST). If you specify a time that falls in DST, specifying 0 (zero) for this parameter causes **mktime** to add one hour to the time. If you specify a time that fall in standard time, specifying any valid monk integer except 0 (zero) causes **mktime** to subtract one hour from the time returned.

Return Value

Monk Time object.

Limitations

On a Windows machine the time must between 1969 Dec 31 4:00:00 PM and 2038 Jan 18 19:14:07. The limitations may be different under other operating systems.

Examples

These examples were created and tested under Windows 2000.

```
(mktime 69 11 31 16 0 0)              => Wed Dec 31 16:00:00 1969

(mktime 70 0 0 16 0 0)                => Wed Dec 31 16:00:00 1969

(mktime 138 0 18 19 14 7)             => Mon Jan 18 19:14:07 2038

(mktime 99 0 1 12 30 33 0)            => Fri Jan 1 12:30:33 1999

(mktime 100 6 1 12 30 33 0)           => Tue Jul 1 13:30:33 2000

(mktime 100 6 1 12 30 33 77777)       => Tue Jul 1 12:30:33 2000

(mktime 100 1 1 12 30 33 88888)       => Tue Feb 1 11:30:33 2000

(mktime 100 1 1 12 30 33 0)           => Tue Feb 1 12:30:33 2000
```

strftime

Syntax

```
(strftime format-spec time )
```

Description

Formats a date/time to user specifications.

Parameters

Name	Type	Description
format-spec	string	A string specifying the format of the date/time. The syntax is the same as accepted by the C library function **strftime**.
time	time object	A time object. You can use the **time** or **mktime** functions to return a time object.

Return Value

Formats the input date according to the format specification and returns the formatted date as a string.

Examples

```
(strftime "%d%b%y" (mktime 70 0 0 16 0 0)) => "31Dec69"

(strftime "%Y%m%d%H%M" (time))             => "200011141330"
```

The **time** function returns the current system time as a Monk time object.

time

Syntax

```
(time)
```

Description

Retrieves the current system time, defined as the number of seconds since midnight, 1 January 1970, Coordinated Universal Time.

Parameters

None.

Return Value

The current system time as a Monk time object.

Example

```
(time) => Thu Apr 15 09:07:33 1999
```

Chapter 17

Interface API Functionality

The Interface API functionality includes:

interface?

Syntax

```
(interface? interface)
```

Description

Checks to see if an interface is available or not. This can only be used with the **load-interface** function. For more information, see **"load-interface" on page 381**.

Parameters

Name	Type	Description
interface	interface	The name of the interface.

Return Value

Returns an interface handle.

Example

```
(interface? bad_interface)     =>     #f

(interface? good-interface)    =>     #t
```

interface-handle

Syntax

```
(interface-handle)
```

Description

Creates a Monk interface handle. This handle allows you to invoke Monk routines from other programs.

Parameters

None.

Return Value

interface
Returns an interface handle to Monk.

Examples

```
(interface-handle)      =>    {MONK_ATOM_TYPE_INTERFACE}
```

invoke

Syntax

```
(invoke obj string [params...])
```

Description

Calls the function contained in the interface handle, passing the function name and parameter values as input.

The **invoke** function is a generic interface to a set of functions within a dll. The interface dll must use the architecture and protocols defined in the **stcextif.h** file, and first be loaded via the **load-interface** function. The resulting handle becomes the first argument of the **invoke** function. The second argument is the name of the function contained in the interface handle. Parameters three and beyond are passed to **invoke** as input arguments to the requested function.

An object that can be called by the invoke function can optionally be called using the object's name alone. For example, the following are equivalent:

```
(invoke my_object my_function)
(my_object my_function)
```

Parameters

Name	Type	Description
object	handle	Interface handle returned by the **load-interface** function.
string	string	Function that is being invoked.
params...	argument	Optional. The parameter(s) specified is dependent upon the argument list in the function being invoked.

Return Value

Return Code	Description
0	Exit status is OK.
1	Invoke of function/procedure call failed.
2	Failed to allocate memory successfully.
3	Unused.
4	Bad parameter to free function.
5	Bad argument to function/procedure call.

load-interface

Syntax

```
(load-interface dll_file [init_fn])
```

Description

Loads a dll. The dll must adhere to the architecture and protocols defined in the **stcextif.h** file.

Parameters

Name	Type	Description
dll_file	string	Path to the dll to be loaded.
init_fn	string	Name of the **init** function to be called. Optional.

Return Value

Returns an interface handle.

Example

```
(define obj (load-interface "sample_ext.dll"))
```

Chapter 18

Debug Procedures

The debug procedures are grouped in two categories:

18.1 Interactive Debug Procedures

The Interactive Debug functions include:

break

Syntax

```
(break)
```

Description

Suspends execution, and permits interaction with the Monk engine within an interactive environment.

Parameters

None.

Return Value

Unspecified.

Examples

```
(define x 5)
(define y 10)
(define z (+ x y))
(break)                  ; break to interact and check
                         ; that variables were set correctly
(display (/ z 2))
```

Additional Information

Within a (break) loop, the following keywords are meaningful:

Keyword	Function
:?	Prints a help message
:cont	Clears all active breaks and resumes processing
:next	Evaluates the next expression, then returns to the “break” state
:pop	Exits the current “break” level and resumes processing. Use this keyword within nested **break** statements.

set-break

Syntax

```
(set-break keyword function keyword1 function1 ... )
```

Description

Sets a breakpoint upon entry or exit of the specified function.

You may set breakpoints for more than one function by specifying additional keyword/function arguments. See **break** on page 383 for more information about breakpoints.

Parameters

Name	Type	Description
keyword	symbol	One of the following: :on-entry :on-exit :all (both :on-entry and :on-exit) :cont (to continue with the execution) :next (go to the next break point) :pop (leave the current break point) :? (help for the Monk debugging)
function	function	The name of a function

Return Value

Unspecified.

Examples

```
(set-break :on-exit my-function)
(my-function)
...function executes...
=> Break :on-exit -- my-function --
=> 1>
```

18.2 Internal Debug Control Procedures

Internal Debug Control functions include:

monk-flag-check? on page 386

monk-flag-clear on page 387

monk-flag-get on page 388

monk-flag-set on page 389

These functions operate on the following debug flags:

all	other-debug
debug-all	path-access-debug
debug-dlls	print-all-failures
file-load-debug	print-all-features
full-stack-debug	rule-trace-debug
make-event-debug	single-stack-debug
map-event-debug	store-last-map-failure
operators-debug	

monk-flag-check?

Syntax

```
(monk-flag-check? flag)
```

Description

Evaluates the flag and checks whether the symbol that represents the flag is active or not.

Parameters

Name	Type	Description
flag	symbol	A well-known Monk flag.

Return Value

Boolean

Returns **#t** if the flag is active or **#f** if the flag is not active.

Examples

```
(monk-flag-clear 'all)
(monk-flag-check? 'map-event-debug)       => #f

(monk-flag-set 'all)
(monk-flag-check? 'map-event-debug)       => #t
```

monk-flag-clear

Syntax

```
(monk-flag-clear flagn)
```

Description

Clears valid Monk flags.

Flagn may be a symbol for a specific monk flag or a 32-bit mask, expressed as an integer.

Parameters

Name	Type	Description
flagn	symbol	Valid Monk flag(s).

Return Value

Boolean

The result of the function is the success or failure of clearing the last flag in the parameter list, which will be **#t** or **#f**.

Example

```
(monk-flag-clear 'debug-all)        => #t
```

monk-flag-get

Syntax

```
(monk-flag-get)
```

Description

Tells you what Monk flags are currently set. You can also use this to return the integer that corresponds to a particular group of set Monk flags, and then use the integer in **monk-set-flag** to set these flags without having to set them individually.

Parameters

None.

Return Value

A integer value whose bits correspond to the Monk flags currently set.

Example

```
(monk-flag-clear ‘all)                        =>#t
(monk-flag-set ‘store-last-map-failure)       =>#t
(monk-flag-get)                               =>2048

(monk-flag-clear ‘all)                        =>#t
(monk-flag-set ‘all)                          =>#t
(monk-flag-get)                               =>793727
```

monk-flag-set

Syntax

```
(monk-flag-set flag [additional flags])
```

Description

Sets the valid Monk flag by using either the symbol name or an integer that corresponds to a particular flag set(s). See **"monk-flag-get" on page 388** for more information.

Parameters

Name	Type	Description
flag	symbol or interger	Valid Monk flag(s). The flags can be referenced individually by symbol name or by the integer that corresponds that flag. You can also reference a set of flags by using the single integer that corresponds to that group of flags.

Return Value

Boolean

The result of the function is the success or failure of setting the last flag in the parameter list, which will be **#t** or **#f.**

Example

```
(monk-flag-set 'make-event-debug)                        =>#t

(monk-flag-set 'make-event-debug 'file-load-debug)       =>#t

(monk-flag-set 2048)                                     =>#t

(monk-flag-set 'make-event-debug 2048)                   =>#t

(monk-flag-set 'all)                                     =>#t
```

Chapter 19

Math-Precision Functions

These functions provide arithmetic operations with a user-definable precision. Arithmetic with large numbers can be done without any loss in the accuracy of the results. To use these functions, load **stc_monkmath.dll** into your environment using the function, **"load-extension" on page 296**.

Important: *The stc_monkmath.dll is not supported on Compaq Tru64 or Linux machines. Therefore the math-precision functions are not supported on this platform.*

The math-precision functions include the following:

mp-absolute-value

Syntax

```
(mp-absolute-value string)
```

Description

Calculates the absolute value of its input argument (quoted number).

Parameters

Name	Type	Description
string	string	Operand.

Return Value

string
The returned string (quoted number) is the absolute value of the input argument.

Example

```
(mp-absolute-value "-123456.789")      => 123456.789
```

mp-add

Syntax

```
(mp-add string1 string2)
```

Description

Adds two multiple precision numbers.

Parameter

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

string
The returned string (quoted number) is the sum of the two numbers input to the function.

Example

```
(mp-add "123.45678" "1.11111")    => 124.56789
```

mp-ceiling

Syntax

```
(mp-ceiling string)
```

Description

Calculates the next higher integer value of the input argument (quoted number).

Parameters

Name	Type	Description
string	string	Operand.

Return Value

string
The returned string (quoted integer) is the next higher integer value of the input argument.

Examples

```
(mp-ceiling "5.4")  => 6
```

```
(mp-ceiling "-5.4") => -5
```

mp-divide

Syntax

```
(mp-divide string1 string2)
```

Description

Divides two multiple precision numbers.

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

string
The returned string (quoted number) is the quotient of the two numbers input to the function.

Example

```
(mp-divide "123.45678" "2.56")    => 48.2253046875
```

mp-even?

Syntax

```
(mp-even? string)
```

Description

Determines whether the input argument (quoted integer) is an even number.

Parameters

Name	Type	Description
string	string	Operand.

Return Value

Boolean

This function returns **#t** if the integer is even. Otherwise, it returns **#f**.

Examples

```
(mp-even? "123456") => #t

(mp-even? "123455") => #f
```

mp-floor

Syntax

```
(mp-floor string)
```

Description

Determines the previous higher integer value of the input argument (quoted number).

Parameters

Name	Type	Description
string	string	Operand.

Return Value

string

The returned string (quoted integer) is the previous higher integer value of the input argument.

Examples

```
(mp-floor "5.4")  => "5"
```

```
(mp-floor "-5.4") => "-6"
```

mp-max

Syntax

```
(mp-max string1 string2)
```

Description

Calculates the maximum value of two multiple precision numbers.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

string

The returned string (quoted number) is the greater of the two numbers input to the function.

Examples

```
(mp-max "123456" "123459")      => "123459"

(mp-max "123.456" "123.459")    => "123.459"
```

mp-min

Syntax

```
(mp-min string1 string2)
```

Description

Calculate the minimum value of two multiple precision numbers.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

string
The returned string (quoted number) is the lesser of the two numbers input to the function.

Examples

```
(mp-min "123456" "123459")      => "123456"

(mp-min "123.456" "123.459")    => "123.456"
```

mp-modulo

Syntax

```
(mp-modulo string1 string2)
```

Description

Calculates the modulo function on two multiple precision integers.

It performs the same calculation as the **mp-remainder** function.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 1.

Return Value

string
The returned string (quoted integer) is the remainder of the integer division of the two numbers input to the function.

Examples

```
(mp-modulo "26" "5")      => "1"

(mp-modulo "45" "3")      => "0"

(mp-modulo "3" "26")      => "3"
```

mp-multiply

Syntax

```
(mp-multiply string1 string2)
```

Description

Multiplies two multiple precision numbers.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

string

The returned string (quoted number) is the product of the two numbers input to the function.

Examples

```
(mp-multiply "123.45678" "1.11111")     => "137.1740628258"

(mp-multiply "45" "3")                  => "135"

(mp-multiply "3" "123.45678")            => "370.37034"
```

mp-negative?

Syntax

```
(mp-negative? string)
```

Description

Determines whether the input argument (quoted number) is a negative number.

Parameters

Name	Type	Description
string	string	Operand.

Return Value

Boolean

This function returns **#t** if the integer is negative. Otherwise, it returns **#f**.

Examples

```
(mp-negative? "-123456")            => #t

(mp-negative? "123455")             => #f

(mp-negative? "3.8")                => #f
```

mp-num-eq

Syntax

```
(mp-num-eq string1 string2)
```

Description

Compares two multiple precision numbers for equality.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

Boolean

This function returns **#t** if the numbers are equal. Otherwise, it returns **#f**.

Examples

```
(mp-num-eq "123.456" "123.456")             => #t

(mp-num-eq "123.455" "123.556")             => #f
```

mp-num-ge

Syntax

```
(mp-num-ge string1 string2)
```

Description

Compares two multiple precision numbers to determine if one is greater than or equal to the other.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

Boolean

This function returns **#t** if *string1* is greater than or equal to *string2*. Otherwise, it returns **#f**.

Examples

```
(mp-num-ge "123.556" "123.556")          => #t

(mp-num-ge "123.656" "123.556")          => #t

(mp-num-ge "123.456" "123.556")          => #f
```

mp-num-gt

Syntax

```
(mp-num-gt string1 string2)
```

Description

Compares two multiple precision numbers to see if one is greater than another.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

Boolean

This function returns **#t** if *string1* is greater than *string2*. Otherwise, it returns **#f**.

Examples

```
(mp-num-gt "123.656" "123.556")          => #t

(mp-num-gt "123.456" "123.556")          => #f
```

mp-num-le

Syntax

```
(mp-num-le string1 string2)
```

Description

Compares two multiple precision numbers to see if one is less than or equal to the other.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

Boolean

This function returns **#t** if *string1* is less than or equal to *string2*. Otherwise, it returns **#f**.

Examples

```
(mp-num-le "123.556" "123.556")        => #t

(mp-num-le "123.456" "123.556")        => #t

(mp-num-le "123.656" "123.556")        => #f
```

mp-num-lt

Syntax

```
(mp-num-lt string1 string2)
```

Description

Compares two multiple precision numbers to determine if one is less than another.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

Boolean

This function returns **#t** if *string1* is less than *string2*. Otherwise, it returns **#f**.

Examples

```
(mp-num-lt "123.556" "123.556")       => #t

(mp-num-lt "123.656" "123.556")       => #f
```

mp-num-ne

Syntax

```
(mp-num-ne string1 string2)
```

Description

Compares two multiple precision numbers to determine if they are not equal to each other.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

Boolean

This function returns **#t** if the numbers are not equal. Otherwise, it returns **#f**.

Examples

```
(mp-num-ne "123.456" "123.556")        => #t

(mp-num-ne "123.456" "123.456")        => #f
```

mp-odd?

Syntax

```
(mp-odd? string)
```

Description

Determines whether the input argument (quoted integer) is an odd number.

Parameters

Name	Type	Description
string	string	Operand.

Return Value

Boolean

This function returns **#t** if the integer is odd. Otherwise, it returns **#f**.

Examples

```
(mp-odd? "123455")    => #t
(mp-odd? "123456")    => #f
```

mp-positive?

Syntax

```
(mp-positive? string)
```

Description

Determines whether the input argument (quoted number) is a positive number.

Parameters

Name	Type	Description
string	string	Operand.

Return Value

Boolean

This function returns **#t** if the number is positive. Otherwise, it returns **#f**.

Examples

```
(mp-positive? "123455")          => #t

(mp-positive? "-123465")         => #f
```

mp-quotient

Syntax

```
(mp-quotient string1 string2)
```

Description

Divides two multiple precision integers.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

string
The returned string (quoted integer) is the integer portion of the quotient.

Example

```
(mp-quotient "20" "7")              => "2"
```

mp-remainder

Syntax

```
(mp-remainder string1 string2)
```

Description

Calculates the remainder after division of two multiple precision integers.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 1.

Return Value

string

The returned string (quoted integer) is the remainder of the integer division of the two numbers input to the function.

Example

```
(mp-remainder "26" "5")            => "1"
```

mp-round

Syntax

```
(mp-round string) or

(mp-round string integer)
```

Description

Rounds off a string argument (quoted number). It also takes a second, optional parameter indicating the rounding depth to the right of the decimal point.

Parameters

Name	Type	Description
string	string	Operand 1.
integer	integer	Operand 2.

Return Value

string

The returned string is the rounded value of the number input to the function.

Examples

```
(mp-round "123.456")                  => "123"

(mp-round "123.567")                  => "124"

(mp-round "123.95" 1)                 => "124.0"

(mp-round "123.45678" 0)              => "123"

(mp-round "123.45678" 1)              => "123.5"

(mp-round "123.45678" 2)              => "123.46"

(mp-round "123.45678" 3)              => "123.457"
```

mp-set-precision

Syntax

```
(mp-set-precision integer)
```

Description

Sets the level of precision for the underlying math engine.

You can enter an integer from 32 to 1024. The default precision for the library is set to 128 bits.

Parameters

Name	Type	Description
number	integer	Number of bits.

Return Value

Unspecified.

Examples

```
(mp-set-precision 256)
```

```
(mp-set-precision 12)
```

mp-subtract

Syntax

```
(mp-subtract string1 string2)
```

Description

Subtracts two multiple precision numbers.

Parameters

Name	Type	Description
string1	string	Operand 1.
string2	string	Operand 2.

Return Value

string
The returned string (quoted number) is the difference of the two numbers input to the function.

Example

```
(mp-subtract "123.45678" "1.11111") => "122.34567"
```

mp-truncate

Syntax

```
(mp-truncate string)
```

Description

Truncates a multiple precision number, by removing the decimal point and any numbers following the decimal point.

Parameters

Name	Type	Description
string	string	Operand 1.

Return Value

string
The returned string (quoted integer) is the integer portion of the number input to the function.

Example

```
(mp-truncate "1234.567")              => "1234"
```

Chapter 20

Monk Library Functions

Monk Library functions are those functions created by SeeBeyond specifically for the user. These functions include:

Basic Library Functions on page 416

Advanced Library Functions on page 460

To use these functions you must load the following directory:

- **<eGate>/client/monk_library**

20.1 Basic Library Functions

allcap?

Syntax

```
(allcap? source)
```

Description

Determines whether or not all ASCII characters are upper case.

Parameters

Name	Type	Description
source	string	The expression to be checked.

Return Value

Boolean

Returns **#t** (true) if all characters in the specified string are upper case. Otherwise, it returns **#f** (false).

Examples

```
(allcap? "ALL CAPS")                => #t

(allcap? "Not All Caps")            => #f
```

capitalize

Syntax

```
(capitalize string)
```

Description

Converts any lower-case letter found in the initial position in the specified string to upper case.

Parameters

Name	Type	Description
string	string	The string to test.

Return Value

string
Containing a copy of the string with any character found in the initial position in the string converted to upper case. If the specified string contains any non-alphanumeric character, a lowercase character following the character will be capitalized.

Examples

```
(capitalize "ABCD")          => "ABCD"

(capitalize "abcd")          => "Abcd"

(capitalize "AB.abcd")       => "AB.Abcd"
```

char-punctuation?

Syntax

```
(char-punctuation? char)
```

Description

Tests the specified character to determine whether or not it is a punctuation character.

Parameters

Name	Type	Description
char	character	The character to be tested.

Return Value

Boolean

Returns **#t** (true) if and only if the specified character is a punctuation character. Otherwise, it returns **#f** (false).

Examples

```
(char-punctuation? #\A)        => #f

(char-punctuation? #\b)        => #f

(char-punctuation? #\3)        => #f

(char-punctuation? #\)         => #f

(char-punctuation? #\;)        => #t
```

char-substitute

Syntax

```
(char-substitute source origchar newchar)
```

Description

Replaces each *origchar* found with a specified *newchar*. A copy of the original *source* with each occurrence of the *origchar* replaced with the *newchar* is returned.

Parameters

Name	Type	Description
source	string	The specified source string.
origchar	character	The character to search for as well as replace.
newchar	character	The replacement character.

Return Value

string
Containing a copy of the original *source* with each occurrence of the original character replaced with the new character.

Example

```
(char-substitute "string a" #\a #\b)       => "string b"
```

char-to-char

Syntax

```
(char-to-char source origchar newchar)
```

Description

Replaces each found *origchar* with a specified *newchar*. Returns a copy of the original *source* with each occurrence of the *origchar* replaced with the *newchar*.

Parameters

Name	Type	Description
source	string	The specified source string.
origchar	character	The character to search for as well as replace.
newchar	character	The replacement character.

Return Value

string

Containing a copy of the original *source* with each occurrence of the original character replaced with the new character. If the *origchar* is not found, the source string is returned.

Example

```
(char-to-char "string a" #\a #\b)     => "string b"
```

conv

Syntax

```
(conv string)
```

Description

Replaces the question mark with a space. This function is a specific example of the more general function, **char-substitute**.

Parameters

Name	Type	Description
string	string	A specified string to test and convert.

Return Value

string
All question marks are replaced by spaces. If no substitution takes place, the original source string is returned.

Example

```
(conv "ab?cd?ef")             > "ab cd ef"
```

count-used-children

Syntax

```
(count-used-children input-path)
```

Description

Retrieves the count of subnodes found on the *input-path* of a node which contains data.

This function can be used to determine the number of subnodes within a event structure if you are performing some type of iterative operation on the structure.

Parameters

Name	Type	Description
input-path	path	The path to be checked for subnodes.

Return Value

number

A count of the subnodes found on the input-path of a node which contain data.

Example

```
(count-used-children ~input%Incoming)   => ; (count of subnodes
containing data)
```

degc->degf

Syntax

```
(degc->degf temp)
```

Description

Converts a temperature from Celsius to Fahrenheit.

Parameters

Name	Type	Description
temp	real number	Temperature in degrees Celsius.

Return Value

number

Returns a number representing the temperature, in Fahrenheit, resulting from the conversion.

Examples

```
(degc->degf 100)      => 212.0

(degc->degf 0.0)      => 32.0
```

degf->degc

Syntax

```
(degf->degc temp)
```

Description

Converts a temperature from Fahrenheit to Celsius.

Parameters

Name	Type	Description
temp	real number	Temperature in degrees Fahrenheit.

Return Value

number

Returns a number representing the temperature, in degrees Celsius, resulting from the conversion.

Examples

```
(degf->degc 212.0)      => 100.0

(degf->degc 32.0)       => 0.0
```

diff-two-dates

Syntax

```
(diff-two-dates date1 date2)
```

Description

diff-two-dates determines the number of days between two standard dates. The function converts the standard dates into a Julian form and subtracts the second date from the first. If the second date is later than the first, the result will be negative.

Parameters

Name	Type	Description
date1	string	First date in format YYYYMMDD.
date2	string	Second date in format YYYYMMDD.

Return Value

integer
Represents the number of days between the two user-specified standard dates. The result may be positive or negative.

Examples

```
(diff-two-dates "19960602" "19960225")       => 98

(diff-two-dates "19960101" "19970101")       => -364
```

display-error

Syntax

```
(display-error data)
```

Description

Writes data from the display statement to the error port.

Parameters

Name	Type	Description
data	string/path	The data to display on the error port;

Return Value

Unspecified.

Example

```
(display-error (string-append "i=" i "\n"))
```

empty-string?

Syntax

```
(empty-string? param)
```

Description

Tests the supplied parameter to determine whether or not it is empty.

Parameters

Name	Type	Description
parm	string	The string to be tested.

Return Value

Boolean

Returns **#t** (true) if the supplied parameter is empty; otherwise, it returns **#f** (false).

Examples

```
(empty-string? "string")          => #f

(empty-string? "")                => #t
```

fail_id

Syntax

```
(fail_id)
```

Description

Aborts the operation.

Parameters

None.

Return Value

None.

Example

```
(fail_id)
```

fail_id_if

Syntax

```
(fail_id_if arg)
```

Description

Aborts the operation if the argument is true.

Parameters

Name	Type	Description
arg	Boolean	The argument to test.

Return Value

None.

Example

```
(fail_id_if (odd? 3))
```

fail_translation

Syntax

```
(fail_translation)
```

Description

Aborts the operation.

Parameters

None.

Return Value

None.

Example

```
(fail_translation)
```

fail_translation_if

Syntax

```
(fail_translation_if arg)
```

Description

Aborts the operation if the argument is true.

Parameters

Name	Type	Description
arg	Boolean	The argument to test.

Return Value

None.

Example

```
(fail_translation_if (odd? 3))
```

find-get-after

Syntax

```
(find-get-after source substring)
```

Description

Searches the specified *source*, looking for the first occurrence of the specified *substring*.

Parameters

Name	Type	Description
source	string	The string to test.
substring	string	The substring to parse.

Return Value

string
If the substring is found, this function returns all characters of the source from the beginning of the first occurrence of the substring to the end of the source.

Boolean
If the substring is not found in source, the function returns #f.

Examples

```
(find-get-after "abcdefghidef" "def")     => "defghidef"

(find-get-after "abcdefghi "jkl")         => #f
```

find-get-before

Syntax

```
(find-get-before source substring)
```

Description

Searches the specified *source* character by character, looking for the specified *substring*.

Parameters

Name	Type	Description
source	string	The string to test.
substring	string	The substring to parse.

Return Value

string

If the substring is found, this function returns all characters of the source from the beginning of source up to but not including the beginning of the first occurrence of the substring.

Boolean

If the substring is not found in the source, the function returns #f.

Examples

```
(find-get-before "abcdefghidef" "def")      => "abc"

(find-get-before "abcdefghi" "jkl")          => #f
```

get-timestamp

Syntax

```
(get-timestamp format)
```

Description

Generates a user-specified timestamp and returns it as a string.

Parameters

Name	Description
format	The specification of the output format. The syntax for the format instruction is documented in **Format Specification** on page 34.

Return Value

string

julian-date?

Syntax

```
(julian-date? date)
```

Description

Determines if the seven-digit date provided in the call is a valid Julian date.

Parameters

Name	Type	Description
date	string	Seven-digit Julian date.

Return Value

Boolean
Returns **#t** (true) if the string is a valid Julian date; otherwise, returns a **#f** (false).

Examples

```
(julian-date? "2444239")     => #t

(julian-date? "244239")      => #f
```

julian->standard

Syntax

```
(julian->standard date)
```

Description

Converts a Julian date to a standard date in the form YYYYMMDD.

Parameters

Name	Type	Description
date	string	Julian date.

Return Value

string
A standard date in the form YYYYMMDD.

Examples

```
(julian->standard "245449")    => "19990927"

(julian->standard "2436078")   => "19570827"
```

leap-year?

Syntax

```
(leap-year? year)
```

Description

Determines if the *year* represents a leap year. The year may be specified as either an integer or as a string value.

Parameters

Name	Type	Description
year	integer/ string	A four-digit integer representing a year.

Return Value

Boolean
Returns **#t** (true) if the integer does represent a leap year; otherwise, returns **#f** (false).

Examples

```
(leap-year? 1990)     => #f

(leap-year? 1996)     => #t
```

map-string

Syntax

```
(map-string function source)
```

Description

Returns a string that is itself the return from a specified Monk function operating on the characters in source. You must specify a Monk character function which also returns a Boolean value as one of its Return Value.

Parameters

Name	Type	Description
function	function	The Monk function to operate on the source string.
source	string	The path or string on which to perform the Monk function.

Return Value

string
The return value of the Monk function operating on the characters in string.

Example

```
(map-string char-upcase "a string")    => "A STRING"
```

not-empty-string?

Syntax

```
(not-empty-string? param)
```

Description

Tests the supplied parameter to determine whether or not it contains data.

Parameters

Name	Type	Description
parm	string	The string to be tested.

Return Value

Boolean

Returns **#t** (true) if the supplied parameter is not empty; otherwise, returns **#f** (false).

Examples

```
(not-empty-string? "string")        => #t

(not-empty-string? "")              => #f
```

standard-date?

Syntax

```
(standard-date? date)
```

Description

Determines if the *date* represents a standard date in the form YYYYMMDD.

Parameters

Name	Type	Description
date	string	A standard date in the form YYYYMMDD.

Return Value

Boolean

Returns **#t** (true) if the supplied string represents a valid standard date of the form YYYYMMDD; otherwise, returns **#f** (false).

Examples

```
(standard-date? "19480115")     => #t

(standard-date? "48015")        => #f
```

standard->julian

Syntax

```
(standard->julian date)
```

Description

Converts a standard date, in the format YYYYMMDD, specified by the *date* parameter, to a Julian date.

Parameters

Name	Type	Description
date	string	A standard date in the form YYYYMMDD.

Return Value

string
Returns the Julian date.

Examples

```
(standard->julian "19480115")      => "2432556"

(standard->julian "18980215")      => "2414716"
```

string-begins-with?

Syntax

```
(string-begins-with? source substring)
```

Description

Determines if the *source* begins with the *substring*.

Parameters

Name	Type	Description
source	string	String to test.
substring	string	Substring to test.

Return Value

Boolean

Returns **#t** (true) if the supplied *source* begins with the supplied *substring*; otherwise, returns **#f** (false).

Examples

```
(string-begins-with? "This is input" "This")     => #t

(string-begins-with? "This is input" "input")    => #f
```

string-contains?

Syntax

```
(string-contains? sourcestring substring)
```

Description

Determines if the *substring* is a member of the *sourcestring*.

Parameters

Name	Type	Description
sourcestring	string	String to test.
substring	string	Substring used to test.

Return Value

Boolean

Returns **#t** (true) if the *substring* appears in the *source string*; otherwise, returns **#f** (false).

Example

```
(string-contains? "lslkjg:jk" "ls")    => #t
```

string-ends-with?

Syntax

```
(string-ends-with? source substring)
```

Description

Determines whether or not the *source* ends with the supplied *substring*.

Parameters

Name	Type	Description
source	string	String to test.
substring	string	Substring used to test.

Return Value

Boolean

Returns **#t** (true) if the *source* ends with the supplied *substring*; otherwise, returns **#f** (true).

Examples

```
(string-ends-with? "This is input" "input")     => #t

(string-ends-with? "This is input" "abc")       => #f
```

string-search-from-left

Syntax

```
(string-search-from-left function source)
```

Description

Searches a string using a specified Monk function to find the first character which matches. It returns the index of the first character in source that causes *function* to return true, or the length of source if no such character exists. You must specify a Monk character function which returns a Boolean value as its return value.

Parameters

Name	Type	Description
function	function	The Monk character function to perform. This function must return a Boolean value.
source	string	The string on which the function performs its character search.

Return Value

integer

Returns the index position of the first character in source that causes *function* to return **#t** (true); otherwise, returns the length of *source* if no such character exists.

Examples

```
(string-search-from-left char-numeric? "345 Elm Ave., #7")  => 0

(string-search-from-left char-upper-case? "345 Elm Ave., #7")  => 4
```

string-search-from-right

Syntax

```
(string-search-from-right function source)
```

Description

Searches a string using a specified Monk function to find the last character which matches. It returns the index of the first character in source that causes *function* to return true, or -1 if no such character exists. You must specify a Monk character function which returns a Boolean value as its return value.

Parameters

Name	Type	Description
function	function	The Monk character function to perform. This function must return a Boolean value.
source	string	The string on which the function performs its character search.

Return Value

integer

Returns the index position of the first character in *source* that causes *function* to return **#t** (true); otherwise, returns -1 if no such character exists.

Examples

```
(string-search-from-right char-numeric? "345 Elm Ave., #7") => 15

(string-search-from-right char-upper-case? "345 Elm Ave., #7") => 8
```

string->ssn

Syntax

```
(string->ssn source)
```

Description

Converts a string of 9 digits to a Social Security number.

Parameters

Name	Type	Description
source	string	A number to convert.

Return Value

Returns one of the following:

string
Returns a string containing the valid social security number in the form nnn-nn-nnnn, where n is a digit between 0 - 9.

Boolean
Returns **#f** (false) if the source is not exactly nine digits in length.

Examples

```
(string-ssn "123456789")          => "123-45-6789"

(string-ssn "91066")              => #f
```

strip-punct

Syntax

```
(strip-punct source)
```

Description

Removes punctuation from the specified *source.*

Parameters

Name	Type	Description
source	string	The string to manipulate.

Return Value

string

Returns a string containing a copy of the *source* with all punctuation removed. If nothing was stripped, the original string is returned.

Example

```
(strip-punct "12 Main St., Apt. 22")   => "12 Main St Apt22"
```

strip-string

Syntax

```
(strip-string function source)
```

Description

Removes all characters from the source string which cause the specified Monk function to evaluate to #t.

Parameters

Name	Type	Description
function char	function	The Monk function to perform on the character.
source	string	The string on which the function performs its character search.

Return Value

string
Returns a string containing a copy of the *source* from which all characters that would cause *function char* to return true have been removed.

Examples

```
(strip-string char-numeric? "345 Elm Ave., #7") => " Elm Ave., #"

(strip-string char-whitespace? "A p p l e")      => "Apple"
```

substring=?

Syntax

```
(substring=? string1 string2 index)
```

Description

Checks if the substring of *string2* starting at the *index* offset is equal to *string1*.

Parameters

Name	Type	Description
string1	string	The string that may be equal to the substring.
string2	string	The string that contains the substring indicated by the index offset.
index	integer	Index offset of the substring.

Return Value

Boolean

Returns **#t** (true) if the substring is equal to *string1*; otherwise, returns **#f** (false).

Examples

```
(substring=? "abc" "xyzabc" 3)        #t

(substring=? "abc" "xyzabc" 0)        #f
```

symbol-table-get

Syntax

```
(symbol-table-get key:string)
```

Description

Queries the symbol table for the specified key string.

Parameters

Name	Type	Description
key:string	symbol	The name of the string.

Return Value

Returns one of the following:

symbol
The symbol for the specified key string.

Boolean
Returns **#f** (false) if the string is not found.

Example

```
(symbol-table-put 'one "1")
(symbol-table-put 'two "2")
(symbol-table-put 'three "3")
(display (symbol-table-get 'three))
```

results in the string “3” being displayed.

symbol-table-put

Syntax

```
(symbol-table-put key:string value)
```

Description

Assigns a string value to a symbol.

Parameters

Name	Type	Description
key:string		The name of the string.
value		The value assigned to the symbol.

Return Value

Returns one of the following:

symbol
The symbol for the specified key string.

Boolean
Returns **#f** (false) if the string is not found.

Examples

```
(symbol-table-put 'one "1")
(symbol-table-put 'two "2")
(symbol-table-put 'three "3")
```

trim-string-left

Syntax

```
(trim-string-left source substring)
```

Description

Removes the specified *substring* from the *source*.

Parameters

Name	Type	Description
source	string	String to test.
substring	string	Substring to remove.

Return Value

string
Returns a copy of the source with all leading occurrences of the substring removed.

Example

```
(trim-string-left "abcdef" "abc")   => "def"
```

trim-string-right

Syntax

```
(trim-string-right source substring)
```

Description

Removes the specified *substring* from the *source*.

Parameters

Name	Type	Description
source	string	The string to test.
substring	string	The substring to remove.

Return Value

string
Returns a copy of the source with all trailing occurrences of the substring removed.

Example

```
(trim-string-right "abcdef" "def")        > "abc"
```

valid-decimal?

Syntax

```
(valid-decimal? number)
```

Description

Tests the *number* to determine if it is a valid decimal number.

Parameters

Name	Type	Description
number	string	The number to test.

Return Value

Boolean

Returns **#t** if the supplied number is a valid decimal number. Otherwise, it returns **#f.**

Examples

```
(valid-decimal? "44.")          => #t

(valid-decimal? "44.0")         => #t

(valid-decimal? "44")           => #f

(valid-decimal? "91066")        => #f
```

valid-integer?

Syntax

```
(valid-integer? number)
```

Description

Tests *number* to determine if it is a valid integer number.

Parameters

Name	Type	Description
number	string	The number to test.

Return Value

Boolean

Returns **#t** if the supplied number is an integer number. Otherwise, it returns **#f**.

Examples

```
(valid-integer? "44")           => #t

(valid-integer? "818")          => #t

(valid-integer? "123.5")        => #f
```

verify-type

Syntax

```
(verify-type checkfunc param)
```

Description

Checks that the argument answers **#t** to the specified Monk function.

If the argument answers **#t**, processing continues. Otherwise an exception condition code is returned which terminates processing. This function is generally used for internal run-time checking. The check function specified must be a Monk function which returns a Boolean value.

Parameters

Name	Type	Description
checkfunc	function	The Monk function to test.
param	integer	The argument to test.

Return Value

None.

Examples

```
(verify-type number? 3)       => ; (continue)

(verify-type number? a)       => ; (exception)
```

20.2 Advanced Library Functions

Before using any of the advanced library functions, you must load them. This is accomplished by either adding **monk_library/advanced** into the monk path or including the line

```
(load-directory "monk_library/advanced")
```

in the Collaboration Rule (.tsc) file where the advanced library function is being used.

The Advanced Library Functions are listed below:

calc-surface-bsa on page 461

calc-surface-gg on page 462

cm->in on page 463

get-2-ssn on page 464

get-3-ssn on page 465

get-4-ssn on page 466

get-apartment on page 467

get-city on page 468

get-first-name on page 469

get-last-name on page 470

get-middle-name on page 471

get-state on page 472

get-street-address on page 473

get-zip on page 474**in->cm** on page 475

lb->oz on page 476

oz->gm on page 477

oz->lb on page 478

valid-phone? on page 479

valid-ssn? on page 480

calc-surface-bsa

Syntax

```
(calc-surface-bsa height weight)
```

Description

Calculates the surface area of a human body in square meters, based on an individual's height, in centimeters, and weight, in kilograms.

The formula for determining the body surface area is: bsa = 0.024265 $(\text{weight})^{0.5378}$ $(\text{height})^{0.3964}$. These calculations are generally performed on newborn babies for determining proper medication doses.

Parameters

Name	Type	Description
height	number or numeric string	Height of the individual in centimeters.
weight	number or numeric string	Weight of the individual in kilograms.

Return Value

number
Returns the calculated body surface area in square meters.

Examples

```
(calc-surface-bsa 144.0 100)              => 2.0708812096829

(calc-surface-bsa "19960101" "19970101")  => -364
```

calc-surface-gg

Syntax

```
(calc-surface-gg height weight)
```

Description

Calculates the surface area of a human body using the Gehan-George formula.

The function takes the height of an individual in centimeters and the weight in kilograms and uses the formula ln (bsa) = 3.75080 + 0.42246 ln (height) + 0.51456 ln (weight), where ln is the $\log_e$ (natural log), to calculate the body surface area in square meters. These calculations are generally performed on newborn babies for determining proper medication doses.

Parameters

Name	Type	Description
height	number or numeric string	Height of the individual in centimeters.
weight	number or numeric string	Weight of the individual in kilograms.

Return Value

number
Returns the calculated body surface area in square meters.

Example

```
(calc-surface-gg 12 12)         => 0.24113634200082
```

cm->in

Syntax

```
(cm->in number)
```

Description

Converts a number from centimeters to inches.

Parameters

Name	Type	Description
number	real number	Number of centimeters.

Return Value

number
Returns the number of inches resulting from the conversion.

Examples

```
(cm->in 2.54)           => 1.0

(cm->in 5.08)           => 2.0
```

get-2-ssn

Syntax

```
(get-2-ssn ssn)
```

Description

Parses the specified social security number and returns the second group of digits.

Parameters

Name	Type	Description
ssn	string	Social security number. A valid ssn string consists of nine digits with a hyphen following the third and fifth digits.

Return Value

number
Returns the second group of digits in a social security number.

Example

```
(get-2-ssn "123-45-6789")    > "45"
```

get-3-ssn

Syntax

```
(get-3-ssn ssn)
```

Description

Parses the specified social security number and returns the first group of digits.

Parameters

Name	Type	Description
ssn	string	Social security number. A valid ssn string consists of nine digits with a hyphen following the third and fifth digits.

Return Value

number

Returns the first group of digits in a social security number.

Example

```
(get-3-ssn "123-45-6789")     => "123"
```

get-4-ssn

Syntax

```
(get-4-ssn ssn)
```

Description

Parses the specified social security number and returns the third group of digits.

Parameters

Name	Type	Description
ssn	string	Social security number. A valid ssn string consists of nine digits with a hyphen following the third and fifth digits.

Return Value

Returns the third group of digits in a social security number.

Example

```
(get-4-ssn "123-45-6789")   => "6789"
```

get-apartment

Syntax

```
(get-apartment address)
```

Description

Returns the apartment information from a string formatted as ADDRESS, APARTMENT, that is, everything after the comma.

Monk does not check the validity of the string, only that a comma exists within it.

Parameters

Name	Type	Description
address	string	Street address.

Return Value

string
Returns a string containing the apartment information from a string formatted as ADDRESS, APARTMENT, that is, everything after the comma.

Examples

```
(get-apartment "12 Main St., Apt. 22")     => "Apt. 22"

(get-apartment "345 Main St., #7")         => "#7"
```

get-city

Syntax

```
(get-city address)
```

Description

Returns the city field from the string formatted as CITY, STATE ZIP.

Monk does not check the validity of the string, only that a comma exists within it.

Parameters

Name	Type	Description
address	string	Address formatted as CITY, STATE ZIP.

Return Value

string
Returns a string containing the city field from the string formatted as CITY, STATE ZIP.

Example

```
(get-city "Arcadia, CA 91066")         => "Arcadia"
```

get-first-name

Syntax

```
(get-first-name name)
```

Description

Returns the first name in a string formatted as LAST, FIRST MIDDLE; that is everything after the first comma and before the next space.

Monk does not check the validity of the string, only that a comma exists within it.

Parameters

Name	Type	Description
name	string	Personal name.

Return Value

string
Returns a string containing the first name in a string formatted as LAST, FIRST MIDDLE; that is, everything after the first comma and before the next space.

Example

```
(get-first-name "Astor, John Jacob")    => "John"
```

get-last-name

Syntax

```
(get-last-name name)
```

Description

Returns the last name in a string formatted as “LAST, FIRST MIDDLE”; that is everything before the comma.

Monk does not check the validity of the string, only that a comma exists within it.

Parameters

Name	Type	Description
name	string	Personal name.

Return Value

string

Returns a string containing the last name in a string formatted as “LAST, FIRST MIDDLE”; that is, everything before the comma.

Example

```
(get-last-name "Astor, John Jacob")   => "Astor"
```

get-middle-name

Syntax

```
(get-middle-name name)
```

Description

Returns the middle name in a string formatted as "LAST, FIRST MIDDLE"; that is everything following the space after the first name.

Monk does not check the validity of the string, only that a comma exists within it. If the data specifies a dual first name, for example Mary Jo Elizabeth Smith, this function will interpret "Jo" as the middle name.

Parameters

Name	Type	Description
name	string	Personal name.

Return Value

string

Returns a string containing the middle name in a string formatted as "LAST, FIRST MIDDLE"; that is, everything after the space.

Example

```
(get-middle-name "Astor, John Jacob")        => "Jacob"
```

get-state

Syntax

```
(get-state address)
```

Description

Returns the state field from the string formatted as “CITY, STATE ZIP.”

Monk does not check the validity of the string, only that a comma exists within it.

Parameters

Name	Type	Description
address	string	Address in the form CITY, STATE, ZIP.

Return Value

string
Returns a string with the state field from the string formatted as CITY,STATE ZIP.

Example

```
(get-state "Arcadia, CA 91066")    => "CA"
```

get-street-address

Syntax

```
(get-street-address address)
```

Description

Returns the address from a string formatted as ADDRESS, APARTMENT, that is everything before the comma. If no comma is specified, it returns the entire string.

Monk does not check the validity of the string, only that a comma exists within it.

Parameters

Name	Type	Description
address	string	Street address.

Return Value

string
Returns a string with the street address from the supplied string, that is, everything before the first comma. If the string does not contain a comma, the function returns the entire string.

Examples

```
(get-street-address "12 Main St., Apt. 22")    => "12 Main St."

(get-street-address "345 Elm Ave., #7")        => "345 Elm Ave."

(get-street-address "345 Elm Ave. #7")         => "345 Elm Ave. #7"
```

get-zip

Syntax

```
(get-zip address)
```

Description

Returns the zip code field from the string formatted as CITY, STATE ZIP.

Monk does not check the validity of the string, only that a comma exists within it.

Parameters

Name	Type	Description
address	string	Address in the form CITY, STATE, ZIP.

Return Value

string
Returns a string with the zip code field from the supplied string formatted as CITY, STATE ZIP.

Example

```
(get-zip "Arcadia, CA 91066")      => "91066"
```

in->cm

Syntax

```
(in->cm number)
```

Description

Converts a number from inches to centimeters.

Parameters

Name	Type	Description
number	real number	Number of inches.

Return Value

number
Returns a number representing the number of centimeters resulting from the conversion.

Examples

```
(in->cm 10.0)      => 25.4

(in->cm 39.4)      => 100.076
```

lb->oz

Syntax

```
(lb->oz number)
```

Description

Converts a number expressed as weight in pounds number and converts this number from pounds to ounces.

Parameters

Name	Type	Description
number	real number	Weight in pounds.

Return Value

number
Returns a number representing the weight in ounces resulting from the conversion.

Examples

```
(lb->oz 2.0)        => 32.0

(lb->oz 6.25)       => 100.0
```

oz->gm

Syntax

```
(oz->gm number)
```

Description

Converts a number which represents weight in ounces to grams.

Parameters

Name	Type	Description
number	real number	Weight in ounces.

Return Value

number

Returns a number representing weight in grams resulting from the conversion.

Examples

```
(oz->gm 0.035)     => .99225
(oz->gm 1.0)       => 28.35
```

oz->lb

Syntax

```
(oz->lb number)
```

Description

Converts a weight in ounces to pounds.

Parameters

Name	Type	Description
number	real number	Weight in ounces.

Return Value

number

Returns a number representing the weight in pounds resulting from the conversion.

Examples

```
(oz->lb 32)        => 2.0

(oz->lb 100)       => 6.25
```

valid-phone?

Syntax

```
(valid-phone? number)
```

Description

Tests the supplied number to determine if it is a valid phone number

A valid phone number is a string of the form NN (NNN) NNN-NNNN, where the first two groups of characters (country code and area code) are both optional, and there can be any number of spaces between the three character groups. Parenthesis are required when entering an area code.

Parameters

Name	Type	Description
number	string	Number to test.

Return Value

Boolean

Returns **#t** if the supplied number is a phone number. Otherwise, it returns **#f**.

Examples

```
(valid-phone? "44(326)323-5909")      => #t

(valid-phone? "(818)445-7000")        => #t

(valid-phone? "123-45-6789")          => #f

(valid-phone? "91066")                => #f
```

valid-ssn?

Syntax

```
(valid-ssn? number)
```

Description

Tests the supplied number to determine if it is a valid social security number.

A valid social security number is a string formatted as DDD-DD-DDDD, where all the D's are digits. Dashes are required between the three groups making up the social security number.

Parameters

Name	Type	Description
number	string	The number to test.

Return Value

Boolean

Returns **#t** if the supplied number is a social security number. Otherwise, it returns **#f**.

Examples

```
(valid-ssn? "123-45-6789")    => #t

(valid-ssn? "91066")          => #f
```

Chapter 21

International Conversion Functions

In the US we have, for the most part, ASCII and to a lesser extent EBCDIC for character encoding. Other countries, on-the-other-hand, have several widely used schemes for encoding characters. For example, in Japan to encode Japanese characters:

- UNIX uses EUC
- WINDOWS uses SJIS
- MAINFRAMES use EBCDICJ and
- EMAIL uses JIS

The Monk engine uses SJIS for encoding Japanese characters in its internal processing. Therefore, it is necessary at times to convert data that uses a different character encoding scheme to SJIS before it can be further processed by the Monk engine. It is also necessary to be able to convert the product of a Monk program back to these other character encoding schemes.

To use these functions you must load the following directories:

- **/eGate/client/monk_library/conversions/japanese**
- **/eGate/client/monk_library/conversions/korean**
- **/eGate/client/monk_library/conversions/UTF8**

The UTF8 Conversion Utility

Additional support for UTF8 conversion is provided through the UTF8 Conversion utility—**utf8convert.exe**. The UTF8 conversion utility is used to convert Collaboration Rules Scripts (.tsc), Event Type Definitions (.ssc), and XML files into UTF8 format.

The UTF8 Conversion utility is located in:

- **/eGate/client/bin/**

UTF8 Conversion utility usage

```
utf8convert -sgbuacghl[123456789] -XM [-i input] [- o output]
```

Table 6 Command Arguments for utf8convert

Parameter	Description
-s	ShiftJIS table
-g	GB2312 file
-b	Big-5 file
-u	UHC file
-a	Arabic file
-c	Cyrillic file
-k	Greek file
-h	Hebrew file
-l[1 2 3 4 5 6 7 8]	Latin file
[-X:]	XML file (option)
[-M:]	MONK (.tsc or .ssc) file (option)
[-i *input*]	Multi-byte file name (option)
[-o *output*]	UTF-8 file name (option)

arabic2utf8

Syntax

```
(arabic2utf8 string)
```

Description

Converts data encoded using the Arabic character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Arabic string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(arabic2utf8 "ABC")
    =>   ABC
```

big52utf8

Syntax

```
(big52utf8 string)
```

Description

Converts data encoded using the Big-5 character encoding scheme to UTF8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Big-5 encoded string to be converted.

Return Value

string
The converted string in UTF8.

Example

```
(big52utf8 "ABC")
    =>   ABC
```

clear-gaiji-table

Syntax

```
(clear-gaiji-table function-name)
```

Description

Removes all Gaiji conversion tables associated with the **function-name**.

Parameters

Name	Type	Description
function-name	string	Function name whose conversion tables are to be removed.

Return Value

None.

Example

```
(clear-gaiji-table "sjis2euc")
```

Additional Information

A table that contained a complete Gaiji conversion would be too large for efficient processing. Consequently, a complete Gaiji conversion is typically broken up into multiple tables. The custom Gaiji conversion functions can use only one table at a time, with the table in use called the active table. The active table and is set by the function **set-gaiji-table**. In order to use a different Gaiji table from the active table, you must first call **clear-gaiji-table** before setting a new active table.

cyrillic2utf8

Syntax

```
(cyrillic2utf8 string)
```

Description

Converts data encoded using the cyrillic character encoding scheme to UTF8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The cyrillic encoded string to be converted.

Return Value

string
The converted string in UTF8.

Example

```
(cyrillic2utf8 "ABC")
    =>  ABC
```

ebcdic2sjis

Syntax

```
(ebcdic2sjis string)
```

Description

Converts data encoded using the EBCDIC-J character encoding scheme to SJIS. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The EBCDIC encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(ebcdic2sjis "ABC")
    =>  íóú
```

ebcdic2sjis_g

Syntax

```
(ebcdic2sjis_g string)
```

Description

Converts data encoded using the EBCDIC character encoding scheme to SJIS using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The EBCDIC encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(ebcdic2sjis_g "ABC")
    =>  íóú
```

ebcdic2uhc

Syntax

```
(ebcdic2uhc string)
```

Description

Converts data encoded using the EBCDIC-J character encoding scheme to UHC. The character type of the converted string is set to :UHC.

Parameters

Name	Type	Description
string	string	The EBCDIC encoded string to be converted.

Return Value

string
The converted string in UHC.

Example

```
(ebcdic2uhc "ABC")
    =>  ¼d
```

ebcdic2uhc_m

Syntax

```
(ebcdic2uhc_m string conversion_mode)
```

Description

Converts single and/or double byte data encoded using the EBCDIC-J character encoding scheme to UHC. The character type of the converted string is set to :UHC.

Parameters

Name	Type	Description
string	string	The EBCDIC encoded string to be converted.
conversion_mode	int	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in UHC.

Example

```
(ebcdic2uhc_m "ABC" 1)
    =>   ABC
```

euc2sjis

Syntax

```
(euc2sjis string)
```

Description

Converts data encoded using the EUC character encoding scheme to SJIS. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The EUC encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(euc2sjis "ABC")
    =>  ABC
```

euc2sjis_g

Syntax

```
(euc2sjis_g string)
```

Description

Converts data encoded using the EUC character encoding scheme to SJIS using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The EUC encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(euc2sjis_g "ABC")
    =>  ABC
```

gb23122utf8

Syntax

```
(gb23122utf8 string)
```

Description

Converts data encoded using the GB2312 character encoding scheme to UTF8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The GB2312 string to be converted.

Return Value

string
The converted string in UTF8.

Example

```
(gb23122utf8 "ABC")
    =>   ABC
```

greek2utf8

Syntax

```
(greek2utf8 string)
```

Description

Converts data encoded using the Greek character encoding scheme to UTF8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Greek encoded string to be converted.

Return Value

string
The converted string in UTF8.

Example

```
(greek2utf8 "ABC")
    =>  ABC
```

hebrew2utf8

Syntax

```
(hebrew2utf8 string)
```

Description

Converts data encoded using the Hebrew character encoding scheme to UTF8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Hebrew encoded string to be converted.

Return Value

string
The converted string in UTF8.

Example

```
(hebrew2utf8 "ABC")
    =>  ABC
```

init-gaiji

Syntax

```
(init-gaiji)
```

Description

Initializes the Gaiji Descriptor in the Monk engine.

Important: *You must call this function before using any of the Japanese Character conversion functions that use custom Gaiji tables.*

Parameters

None.

Return Value

None.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
```

init-utf8gaiji

Syntax

```
(init-utf8gaiji)
```

Description

Initializes the UTF8-Gaiji Descriptor in the Monk engine.

Important: *You must call this function before using any of the UTF8 Japanese Character conversion functions that use custom Gaiji tables.*

Parameters

None.

Return Value

None.

Example

```
(init-utf8gaiji)
(set-utf8gaiji-table "utf8big5")
```

jef2sjis

Syntax

```
(jef2sjis string)
```

Description

Converts data encoded using the JEF character encoding scheme to SJIS. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JEF encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(jef2sjis "ABC")
    =>  ABC
```

jef2sjis_g

Syntax

```
(jef2sjis_g string)
```

Description

Converts data encoded using the JEF character encoding scheme to SJIS using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JEF encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(jef2sjis_g "ABC")
    =>  ABC
```

jef2sjis_m

Syntax

```
(jef2sjis_m string)
```

Description

Converts single and/or double byte data encoded using the JEF character encoding scheme to SJIS. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JEF encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(jef2sjis_m "ABC")
    =>  ABC
```

jef2sjis_m_g

Syntax

```
(jef2sjis_m_g string)
```

Description

Converts single and/or double byte JEF string using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JEF encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(jef2sjis_m_g "ABC")
    =>  ABC
```

jef2sjis_p

Syntax

```
(jef2sjis_p string conversion_mode)
```

Description

Converts data encoded using the JEF character encoding scheme to SJIS using a hexadecimal KI (Kanji In) code. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JEF encoded string to be converted.
conversion_mode	int	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in SJIS.

Example

```
(jef2sjis_p "ABC" 2)
    =>  ABC
```

jef2sjis_p_g

Syntax

```
(jef2sjis_p_g string conversion_mode)
```

Description

Converts data encoded using the JEF character encoding scheme to SJIS using a hexadecimal KI (Kanji In) code and a user-defined custom Gaiji conversion table. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JEF encoded string to be converted.
conversion_mode	int	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string

The converted string in SJIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(jef2sjis_p_g "ABC" 2)
    =>  ABC
```

jipse2sjis

Syntax

```
(jipse2sjis string type)
```

Description

Converts data encoded using the JIPSE character encoding scheme to SJIS. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JIPSE encoded string to be converted.
type	integer	Describes the type of characters in the string being converted. One of the following: ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in SJIS.

Example

```
(jipse2sjis "ABC" 0)
    =>  íóú
```

jipse2sjis_g

Syntax

```
(jipse2sjis_g string type)
```

Description

Converts data encoded using the JIPSE character encoding scheme to SJIS using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JIPSE encoded string to be converted.
type	integer	Describes the type of characters in the string being converted. One of the following: ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(jipse2sjis_g "ABC" 0)
    =>  íóú
```

jis2sjis

Syntax

```
(jis2sjis string)
```

Description

Converts data encoded using the JIS character encoding scheme to SJIS. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JIS encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(jis2sjis "ABC")
    =>   ABC
```

jis2sjis_g

Syntax

```
(jis2sjis_g string)
```

Description

Converts data encoded using the JIS character encoding scheme to SJIS using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The JIS encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(jis2sjis_g "ABC")
    =>  ABC
```

keis2sjis

Syntax

```
(keis2sjis string type)
```

Description

Converts data encoded using the KEIS 83 character encoding scheme to SJIS. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The KEIS 83 encoded string to be converted.
type	integer	Describes the type of characters in the string being converted. One of the following: ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in SJIS.

Example

```
(keis2sjis "ABC" 0)
    =>  íóú
```

keis2sjis_g

Syntax

```
(keis2sjis_g string type)
```

Description

Converts data encoded using the KEIS 83 character encoding scheme to SJIS using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The KEIS 83 encoded string to be converted.
type	integer	Describes the type of characters in the string being converted. One of the following: ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(keis2sjis_g "ABC" 0)
    =>  íóú
```

latin12utf8

Syntax

```
(latin12uft8 string)
```

Description

Converts data encoded using the Latin 1 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 1 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin12uft8 "ABC")
    =>  ABC
```

latin22utf8

Syntax

```
(latin22uft8 string)
```

Description

Converts data encoded using the Latin 2 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 2 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin22uft8 "ABC")
    =>   ABC
```

latin32utf8

Syntax

```
(latin32uft8 string)
```

Description

Converts data encoded using the Latin 3 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 3 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin32uft8 "ABC")
    =>  ABC
```

latin42utf8

Syntax

```
(latin42uft8 string)
```

Description

Converts data encoded using the Latin 4 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 4 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin42uft8 "ABC")
    =>  ABC
```

latin52utf8

Syntax

```
(latin52uft8 string)
```

Description

Converts data encoded using the Latin 5 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 5 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin52uft8 "ABC")
    =>  ABC
```

latin62utf8

Syntax

```
(latin62uft8 string)
```

Description

Converts data encoded using the Latin 6 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 6 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin62uft8 "ABC")
    =>  ABC
```

latin72utf8

Syntax

```
(latin72uft8 string)
```

Description

Converts data encoded using the Latin 7 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 7 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin72uft8 "ABC")
    =>   ABC
```

latin82utf8

Syntax

```
(latin82uft8 string)
```

Description

Converts data encoded using the Latin 8 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 8 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin82uft8 "ABC")
    =>   ABC
```

latin92utf8

Syntax

```
(latin92uft8 string)
```

Description

Converts data encoded using the Latin 9 character encoding scheme to UFT8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The Latin 9 string to be converted.

Return Value

string
The converted string in UFT8.

Example

```
(latin92uft8 "ABC")
    =>   ABC
```

set-gaiji-table

Syntax

```
(set-gaiji-table function-name table-file-name)
```

Description

Sets the **table-file-name** as a Gaiji table for the conversion function **function-name**.

Parameters

Name	Type	Description
function-name	string	Name of the function.
table-file-name	string	Name of the file containing the Gaiji conversion table.

Return Value

None.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
```

Additional Information

Gaiji Table Format:

```
# is comment
# Source Code          Destination Code
0x1234                 0x3456
0x1235                 0x3457
```

set-utf8gaiji-table

Syntax

```
(set-utf8gaiji-table function-name table-file-name)
```

Description

Sets the **table-file-name** as a UTF8 Gaiji table for the conversion function **function-name**.

Parameters

Name	Type	Description
function-name	string	Name of the function.
table-file-name	string	Name of the file containing the UTF8 Gaiji conversion table.

Return Value

None.

Example

```
(init-utf8gaiji)
(set-utf8gaiji-table "sjis2euc" "convert1")
```

Additional Information

UTF8 Gaiji Table Format:

```
# is comment
# Source Code      Destination Code
0x1234             0x3456
0x1235             0x3457
```

sjis2ebcdic

Syntax

```
(sjis2ebcdic string)
```

Description

Converts an SJIS string into EBCDIC-J, then sets its type as :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.

Return Value

string
The converted string in EBCDIC-J.

Example

```
(sjis2ebcdic "íóú")
    =>   ABC
```

sjis2ebcdic_g

Syntax

```
(sjis2ebcdic_g string)
```

Description

Converts an SJIS string into EBCDIC-J using a user-defined custom Gaiji table associated with this function. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.

Return Value

string
The converted string in EBCDIC-J.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(sjis2ebcidic_g "íóú")
    =>  ABC
```

sjis2euc

Syntax

```
(sjis2euc string)
```

Description

Converts an SJIS string into EUC, then sets its type as :EUC.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.

Return Value

string
The converted string in EUC.

Example

```
(sjis2euc "ABC")
    =>   ABC
```

sjis2euc_g

Syntax

```
(sjis2euc_g string)
```

Description

Converts an SJIS string into EUC using a user-defined custom Gaiji table associated with this function. The character type of the converted string is set to :EUC.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.

Return Value

string
The converted string in EUC.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(sjis2euc_g "ABC")
    =>  ABC
```

sjis2jef

Syntax

```
(sjis2jef string)
```

Description

Converts data encoded using the SJIS character encoding scheme to JEF. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS encoded string to be converted

Return Value

string
The converted string in JEF

Example

```
(sjis2jef "ABC")
    =>  ABC
```

sjis2jef_g

Syntax

```
(sjis2jef_g string)
```

Description

Converts data encoded using the SJIS character encoding scheme to JEF using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS encoded string to be converted.

Return Value

string
The converted string in JEF.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(sjis2jef_g "ABC")
    =>  ABC
```

sjis2jef_m

Syntax

```
(sjis2jef_m string)
```

Description

Converts single and/or double byte data encoded using the SJIS character encoding scheme to JEF. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS encoded string to be converted.

Return Value

string
The converted string in JEF.

Example

```
(sjis2jef_m "ABC")
    =>  ABC
```

sjis2jef_m_g

Syntax

```
(sjis2jef_m_g string)
```

Description

Converts single and/or double byte SJIS string using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(sjis2jef_m_g "ABC")
    =>  ABC
```

sjis2jef_p

Syntax

```
(sjis2jef_p string conversion_mode)
```

Description

Converts data encoded using the SJIS character encoding scheme to JEF using a hexadecimal KI (Kanji In) code. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS encoded string to be converted.
conversion_mode	int	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string

The converted string in JEF.

Example

```
(sjis2jef_p "ABC" 2)
    =>  ABC
```

sjis2jef_p_g

Syntax

```
(sjis2jef_p_g string conversion_mode)
```

Description

Converts data encoded using the SJIS character encoding scheme to JEF using a hexadecimal KI (Kanji In) code and a user-defined custom Gaiji conversion table. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS encoded string to be converted.
conversion_mode	int	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in JEF.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(sjis2jef_p_g "ABC" 1)
    =>  ABC
```

sjis2jipse

Syntax

```
(sjis2jipse string type)
```

Description

Converts an SJIS string into JIPSE, then sets its type as :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.
type	integer	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in JIPSE.

Example

```
(sjis2jipse "íóú" 0)
    =>  ABC
```

sjis2jipse_g

Syntax

```
(sjis2jipse_g string type)
```

Description

Converts an SJIS string into JIPSE using a user-defined custom Gaiji table associated with this function. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.
type	int	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in JIPSE.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(sjis2jipse_g "íóú" 0)
    =>  ABC
```

sjis2jis

Syntax

```
(sjis2jis string)
```

Description

Converts an SJIS string into JIS, then sets its type as :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.

Return Value

string
The converted string in JIS.

Example

```
(sjis2jipse "ABC")
    =>  ABC
```

sjis2jis_g

Syntax

```
(sjis2jis_g string type)
```

Description

Converts an SJIS string into JIS using a user-defined custom Gaiji table associated with this function. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.
type	int	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in JIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(sjis2jis_g "ABC" 0)
    =>   ABC
```

sjis2keis

Syntax

```
(sjis2keis string type)
```

Description

Converts an SJIS string into KEIS 83, then sets its type as :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.
type	integer	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string
The converted string in KEIS 83.

Example

```
(sjis2keis "íóú" 0)
    =>   ABC
```

sjis2keis_g

Syntax

```
(sjis2keis_g string type)
```

Description

Converts data encoded using the SJIS character encoding scheme to KEIS 83 using a user-defined custom Gaiji table associated with this function. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The SJIS string to be converted.
type	integer	Indicates the number of bytes in the string to be converted. ■ **0** = Mixed single and/or double byte. ■ **1** = Single byte character. ■ **2** = Double byte character.

Return Value

string
The converted string in KEIS 83.

Example

```
(sjis2keis_g "íóú" 0)
    =>  ABC
```

sjis2sjis

Syntax

```
(sjis2sjis string)
```

Description

Sets the type of string to :SJIS.

Parameters

Name	Type	Description
string	string	String that will be set as :SJIS.

Return Value

string
The converted string in SJIS.

Example

```
(sjis2sjis "ABC")
    =>   ABC
```

sjis2utf8

Syntax

```
(sjis2utf8 string)
```

Description

Converts data encoded using the SJIS character encoding scheme to UTF8. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The SJIS encoded string to be converted.

Return Value

string
The converted string in UTF8.

Example

```
(sjis2utf8 "ABC")
    =>  ABC
```

sjis2utf8_g

Syntax

```
(sjis2utf8_g string)
```

Description

Converts data encoded using the SJIS character encoding scheme to UTF8 using a user-defined custom Gaiji table associated with this function. The character type of the converted string is set to :UTF8.

Parameters

Name	Type	Description
string	string	The SJIS encoded string to be converted.

Return Value

string
The converted string in UTF8.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(sjis2utf8_g "ABC")
    =>  ABC
```

uhc2ebcdic

Syntax

```
(uhc2ebcdic string)
```

Description

Converts data encoded using the UHC character encoding scheme to EBCDIC. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UHC encoded string to be converted.

Return Value

string
The converted string in EBCDIC.

Example

```
(uhc2ebcdic "ABC")
    =>   B-B-B+
```

uhc2ebcdic_m

```
(uhc2ebcdic_m string conversion_mode)
```

Description

Converts single and/or double byte data encoded using the UHC character encoding scheme to EBCDIC. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UHC encoded string to be converted.
conversion_mode	int	Indicates the number of bytes in the string to be converted. ▪ **0** = Mixed single and/or double byte. ▪ **1** = Single byte character. ▪ **2** = Double byte character.

Return Value

string

The converted string in EBCDIC.

Example

```
(uhc2ebcdic_m "ABC" 1)
    =>  ABC
```

uhc2ksc

Syntax

```
(uhc2ksc string)
```

Description

Converts data encoded using the UHC character encoding scheme to KSC. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UHC encoded string to be converted.

Return Value

string
The converted string in KSC.

Example

```
(uhc2ksc "ABC")
    =>  ABC
```

uhc2ksc_m

Syntax

```
(uhc2ksc_m string)
```

Description

Converts single and/or double byte data encoded using the UHC character encoding scheme to KSC. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UHC encoded string to be converted.

Return Value

string
The converted string in KSC.

Example

```
(uhc2ksc_m "ABC")
    =>  ABC
```

uhc2uhc

Syntax

```
(uhc2uhc string)
```

Description

Sets the type of the string to :UHC.

Parameters

Name	Type	Description
string	string	String that will be set as :UHC.

Return Value

string
The converted string in UHC.

Example

```
(uhc2uhc "ABC")
    =>   ABC
```

uhc2utf8

Syntax

```
(uhc2utf8 string)
```

Description

Converts data encoded using the UHC character encoding scheme to UTF8. The character type of the converted string is set to :UTF8

Parameters

Name	Type	Description
string	string	The UHC encoded string to be converted.

Return Value

string
The converted string in UHC.

Example

```
(uhc2utf8 "ABC")
    =>  ABC
```

utf82arabic

Syntax

```
(utf82arabic string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Arabic. The character type of the converted string is set to :1Byte

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string.

Example

```
(utf82arabic "ABC")
    =>  ABC
```

utf82big5

Syntax

```
(utf82big5 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to BIG5. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Big-5.

Example

```
(utf82big5 "ABC")
    =>  ABC
```

utf82cyrillic

Syntax

```
(utf82cyrillic string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Cyrillic. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Cyrillic.

Example

```
(utf82cyrillic "ABC")
    =>  ABC
```

utf82gb2312

Syntax

```
(utf82gb2312 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to GB2312. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in GB2312.

Example

```
(utf82gb2312 "ABC")
    =>  ABC
```

utf82greek

Syntax

```
(utf82greek string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Greek. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string.

Example

```
(utf82greek "ABC")
    =>  ABC
```

utf82hebrew

Syntax

```
(utf82hebrew string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Hebrew. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Hebrew.

Example

```
(utf82hebrew "ABC")
    =>   ABC
```

utf82latin1

Syntax

```
(utf82latin1 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin1. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin1.

Example

```
(utf82latin1 "ABC")
    =>  ABC
```

utf82latin2

Syntax

```
(utf82latin2 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin2 The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin2.

Example

```
(utf82latin2 "ABC")
    =>   ABC
```

utf82latin3

Syntax

```
(utf82latin2 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin2. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin2.

Example

```
(utf82latin3 "ABC")
    =>   ABC
```

utf82latin4

Syntax

```
(utf82latin4 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin4. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin4.

Example

```
(utf82latin4 "ABC")
    =>   ABC
```

utf82latin5

Syntax

```
(utf82latin5 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin5. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin5.

Example

```
(utf82latin5 "ABC")
    =>   ABC
```

utf82latin6

Syntax

```
(utf82latin6 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin6. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin6.

Example

```
(utf82latin6 "ABC")
    =>  ABC
```

utf82latin7

Syntax

```
(utf82latin7 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin7. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin7.

Example

```
(utf82latin7 "ABC")
    =>   ABC
```

utf82latin8

Syntax

```
(utf82latin8 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin8. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin8.

Example

```
(utf82latin8 "ABC")
    =>  ABC
```

utf82latin9

Syntax

```
(utf82latin9 string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to Latin9. The character type of the converted string is set to :1Byte.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in Latin9.

Example

```
(utf82latin9 "ABC")
    =>   ABC
```

utf82sjis

Syntax

```
(utf82sjis string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to SJIS. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in SJIS.

Example

```
(utf82sjis "ABC")
    =>  ABC
```

utf82sjis_g

Syntax

```
(utf82sjis_g string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to SJIS using a user-defined custom Gaiji conversion table associated with this function. The character type of the converted string is set to :SJIS.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted

Return Value

string
The converted string in SJIS.

Example

```
(init-gaiji)
(set-gaiji-table "sjis2euc" "convert1")
(utf82sjis_g "ABC")
    =>  ABC
```

utf82uhc

Syntax

```
(utf82uhc string)
```

Description

Converts data encoded using the UTF8 character encoding scheme to UHC. The character type of the converted string is set to :UHC.

Parameters

Name	Type	Description
string	string	The UTF8 encoded string to be converted.

Return Value

string
The converted string in UHC.

Example

```
(utf82uhc "ABC")
    =>  ABC
```

utf82utf8

Syntax

```
(utf82utf8 string)
```

Description

Sets the type of the string to :UTF8.

Parameters

Name	Type	Description
string	string	String that will be set as :UTF8.

Return Value

string
The converted string in UTF8.

Example

```
(utf82utf8 "ABC")
    =>   ABC
```

Chapter 22

e*Gate Extensions to Monk

This chapter explains the Monk functions that extend the Monk environment. Instructions in each section discuss how to load the extensions into the Monk environment. These functions include:

22.1 Queue Service Access

The queue service access functions allow interaction between the Monk environment and the e*Gate system. Specifically, they provide increased control over the event flow. These functions are automatically loaded when you use either the Monk or the Monk ID Collaboration Service. The queue service access functions are:

iq-get

Syntax

```
(iq-get input-topic event-handle)
```

Description

Gets an Event of the type specified from an IQ, if an Event of that type is available.

If an Event is returned, the queuing service marks the Event as accessed for the subscriber under which **iq-get** was called. If the caller provides an input-topic (Event Type) and 0 for the event-handle, **iq-get** returns the next Event available for that Event Type. If the caller provides an input-topic and a valid event-handle, the Event associated with the specified event-handle is returned.

iq-get can retrieve an Event from any IQ included on the list of topics returned by **iq-input-topics**.

For this function to operate properly it must be run within an environment that provides the correct Event handle—such as within a translation used by a Collaboration in an e*Gate schema.

Note: *When using this function with stctrans, you cannot use 0 for the handle—you must use a valid handle instead.*

Parameters

Name	Type	Description
input-topic	string	Name of the Event Type to get.
handle	Event handle	One of the following: 0—The next Event in the IQ. Event handle—The Event associated with the Event handle. Used to access headers for Events if multiple gets are called on the same Event Type.

Return Value

Returns one of the following:

vector
If an Event is available, **iq-get** returns a vector containing the Event and the Event handle.

Boolean
If no Event is available, **iq-get** returns **#f** (false).

Example

```
(iq-get "input" 0)
```

Additional Information

To run using stctrans, a valid handle must be passed.

iq-get-header

Syntax

```
(iq-get-header handle)
```

Description

Returns the event header for the input event. The *handle* (string) is used to access headers for events if multiple gets are called on the same event type.

Parameters

Name	Type	Description
handle	string	Used to access headers for events if multiple gets are called on the same event type.

Return Value

If there is no header for this event handle the function returns a Boolean **#f**. Call failure will throw an exception. The return is a vector containing the following information for the input event referred to:

Subscriber	(vector)
Publisher	(string)
Priority	(number)
MajorSeqNumber	(number)
MinorSeqNumber	(number

Example

```
; get the initial message header
  (define vMessageHeader (iq-get-header szMessageHandle))
```

Important: *This Monk function is not supported by JMS IQs.*

iq-initial-handle

Syntax

```
(iq-initial-handle)
```

Description

Returns the queue handle of the event which invoked the current event collaboration or identification process.

Parameters

None.

Return Value

A valid event handle.

Example

```
;get the initial message handle
 (define szMessageHandle (iq-initial-handle))
```

iq-initial-topic

Syntax

```
(iq-initial-topic)
```

Description

Returns a string containing the event topic which invoked the current event collaboration or identification process.

Parameters

None.

Return Value

string
event topic

Example

```
;get the initial message type
 (define szMessageType (iq-initial-topic))
 (display (string-append "Message type of initiating message: "
szMessageType "\n"))
```

iq-input-topics

Syntax

```
(iq-input-topics)
```

Description

Returns a vector of strings, containing the names of the event types the component is configured to subscribe to.

Parameters

None.

Return Value

Call failure will throw an exception. Otherwise, a vector containing all input event types.

Example

```
; get the input Event Types
(define vEventTypes (iq-input-topics))
(display "Input Event Types: ")
(display vEventTypes)
(newline)
```

iq-mark-unusable

Syntax

```
(iq-mark-unusable message-handle)
```

Description

Marks the message as unusable. The *message-handle* can be obtained from **iq-initial-handle** or **iq-peek** functions.

Parameters

Name	Type	Description
handle	string	Used to access headers for events if multiple gets are called on the same event type.

Return Value

Boolean
#t or **#f**

Example

```
; mark the Event unusable
 (define szEventType (iq-mark-unusable szEventHandle))
```

Important: *This Monk function is not supported by JMS IQs.*

iq-output-topics

Syntax

```
(iq-output-topics)
```

Description

Returns a vector of strings, containing the names of the output event types the component is configured to publish.

Parameters

None.

Return Value

Call failure will throw an exception. Otherwise, a vector of event types.

Example

```
; get the output Event Types
(define vEventTypes (iq-output-topics))
(display "Output Event Types: ")
(display vEventTypes)
(newline)
```

iq-peek

Syntax

```
(iq-peek input-topic handle)
```

Description

Accesses additional events from the input queues without changing the event state in the queuing service. The transformation function can get from any input queue included on the list of topics in the (**iq-input-topics**) vector.

Parameters

Name	Type	Description
input-topic	string	Name of the event type to get.
handle	valid handle or 0	Used to access headers for events if multiple gets are called on the same event type.

Return Value

The call returns a vector containing the next event and the event handle if a event is available, a Boolean if no data is available, and it throws an exception if the call failed for any other reason. If the caller provides an input topic name and a handle containing the number 0, the call will return the next event available for that input topic. If the caller provides a valid event handle and input topic, next event available relative to the supplied event handle is returned.

Example

```
(display "Performing peek operations on input queues:\n")
(do
  ((i 0 (+ i 1)))
  ((= i n_in))
  (define vMessageAndHandle
    (iq-peek (vector-ref vInputMessageTypes i) 0))
)
```

Important: *This Monk function is not supported by JMS IQs.*

iq-put

Syntax

```
(iq-put output-event-type event input-event-type priority
major-seq-num minor-seq-num)
```

Description

Places an Event on the output queue but does not commit it to the queue until the Monk transformation or identification function returns successfully.

If the Monk function is operating under the Monk Collaboration service and the transformation is only generating a single Event, it does not have to make an explicit call to **iq-put** to forward the Event to the queuing system.

You should include this call if a Monk Collaboration generates more than one output Event.

The Monk Collaboration service enqueues the returned string to the default Event Type vector. The output Event Type and input Event Type must be from the list of configured Event Types that the component is able to receive and produce. The input Event Type is included to help maintain the history of the Event as it passes through the system.

All Events of lower priority level are dequeued before any Events of a higher priority level. Priority zero Events are dequeued first. In typical usage, all calls to this function will be made with the same priority level.

Setting the priority to 0 using the **iq-put** function designates that the caller is letting the system assign the default priority as entered in the Schema Designer. If the default priority has been defined as 0 in the Schema Designer, then messages published by **iq-put** will have a priority of 0.

Parameters

Name	Type	Description
output-event-type	string	Name of the Event Type to which to publish.
event	string	The Event to publish.
input-event-type	list	List of input Event Types which were used to create this Event.
priority	number	Priority to assign to the output Event. Default is 0.
major-seq-num	number	Major sequence number to assign.
minor-seq-num	number	Minor sequence number to assign. An entry of 0 defaults major and minor sequence numbers.

Return Value

Boolean
Returns **#t** if the Event was successfully placed on the queue.

Throws

Exception-Generic

Examples

```
(try
  (iq-put
    "OutEmpEvent"
    szMessage
    (list "EmpData")
    2 0 0
  )
  (catch
    ((Exception-Generic)
      (display "Exception Raised: exception category: ")
      (display (number->string (exception-category))) (newline)
      (display "exception symbol: ")
      (display (symbol->string (exception-symbol))) (newline)
      (display "exception string: ")
      (display (exception-string)) (newline)
    )
  )
)
```

This example queues an Event of type “OutEmpEvent” to the queue. This Event must be one of the Events that the Collaboration publishes to.

The queued Event depends upon an input Event type called “EmpData”. The input Event must be one of the Events that the Collaboration subscribes to. Check the Collaboration details in the e*Gate GUI.

Enclosing the **iq-put** function in a **try...catch** clause is the normal way to handle possible queue errors. This example simply displays information to the log file, but you may want to include more robust error recovery in the **catch** clause.

Additional Information

The **iq-put** function is not supported by the Monk Test Console. For testing purposes, the following solution is suggested:

```
(define iq-put
  (lambda (p1 p2 p3 p4 p5 p6)
    (display (string-append "iq-put: EventTYPE|" p1
              "|EventCONTENT|"p2))
    (newline)
    ""
  )
)
```

The sample script shown above can be used as a *dependency file* when testing a collaboration that uses **iq-put**.

iq-put-411

Syntax

```
(iq-put-411 output-event-type event input-event-type priority
major-seq-num minor-seq-num)
```

Description

This function replicates the behavior of the **iq-put** function in e*Gate version 4.1.1. In that version of e*Gate, **iq-put** returned **#t** or **#f**. The current version of **iq-put** returns **#t** or an exception. The **iq-put-411** function was created to provide backward compatibility for code written for e*Gate 4.1.1.

The **iq-put-411** function makes a call to **iq-put**. If an exception is thrown, it will catch the exception and return **#f**. The exception will be noted in the log file.

For more information on the **iq-put** function, see **iq-put** on page 576.

Parameters

Name	Type	Description
output-event-type	string	Name of the Event Type to which to publish.
event	string	The Event to publish.
input-event-type	list	List of input Event Types which were used to create this Event.
priority	number	Priority to assign to the output Event. Default is 0.
major-seq-num	number	Major sequence number to assign.
minor-seq-num	number	Minor sequence number to assign. An entry of 0 defaults major and minor sequence numbers.

Return Value

Boolean

Returns **#t** if the Event was successfully placed on the queue. Otherwise, it returns **#f**.

Examples

```
(if (iq-put-411 "et_middle" (get ~input%GenericInEvent) (list (iq-
initial-topic)) 5 0 0)
  (begin
    (display "The iq-put-411 returned #t") (newline)
  )
  (begin
    (display "The iq-put-411 returned #f") (newline)
  )
)
```

22.2 e*Way Functions

The following functions are available to all e*Ways based on the Extension Kit (the Generic Monk based e*Ways) in the external Monk environment, that is, the Monk environment that supports the e*Way's configuration file.

Important: *These functions are not available to the internal Monk environment, that is, the Monk environment that supports the e*Way's Collaborations. See a Generic Monk based e*Way User's Guide for more information on the differences between the two Monk environments.*

event-send-to-egate

Syntax

```
(event-send-to-egate string)
```

Description

Sends data that the e*Way has already received from the external system into the e*Gate system as an Event.

Parameters

Name	Type	Description
string	string	The data to be sent to the e*Gate system

Return Value

Boolean

Returns **#t** if the data is sent successfully. Otherwise, returns **#f**.

Notes

This function can be called by any e*Way function when it is necessary to send data to the e*Gate system in a blocking fashion.

get-logical-name

Syntax

```
(get-logical-name)
```

Description

Retrieves the logical name of the e*Way.

Parameters

None.

Return Value

string
Returns the name of the e*Way (as defined by the Schema Designer).

Throws

None.

Additional Information

The **get-logical-name** function cannot be loaded externally from a .dll, because it is already loaded into the external thread by the e*Way and/or BOB executable.

There is an equivalent function for use with Collaborations in the internal Monk environment. It is named **collab-get-logical-name** and is available when you load the **stc_monkext.dll**.

send-external-down

Syntax

```
(send-external-down)
```

Description

Instructs the e*Way that the connection to the external system is down.

Parameters

None.

Return Value

None.

send-external-up

Syntax

```
(send-external-up)
```

Description

Instructs the e*Way that the connection to the external system is up.

Parameters

None.

Return Value

None.

shutdown-request

Syntax

```
(shutdown-request)
```

Description

Completes the e*Gate shutdown procedure that was initiated by the Control Broker but was interrupted by returning a non-null value within the Generic e*Way Shutdown Command Notification Function.

Once this function is called, shutdown proceeds immediately.

Once interrupted, the e*Way's shutdown cannot proceed until this Monk function is called. If you do interrupt an e*Way shutdown, we recommend that you complete the process in a timely fashion.

Parameters

None.

Return Value

None.

start-schedule

Syntax

```
(start-schedule)
```

Description

Requests that the e*Way execute the “Exchange Data with External” function specified within the e*Way’s configuration file. Does not affect any defined schedules.

Parameters

None.

Return Value

None.

stop-schedule

Syntax

```
(stop-schedule)
```

Description

Requests that the e*Way halt execution of the “Exchange Data with External” function specified within the e*Way’s configuration file.

Execution will be stopped when the e*Way concludes any open transaction. Does not affect any defined schedules, and does not halt the e*Way process itself.

Parameters

None.

Return Value

None.

22.3 Monk Extension Functions

The Monk Extension Functions are accessed by loading **stc_monkext.dll**. The Monk Extension functions include:

collab-get-logical-name

Syntax

```
(collab-get-logical-name)
```

Description

Retrieves the logical name of the e*Way.

Parameters

None.

Return Value

string
Returns the name of the e*Way (as defined by the Schema Designer).

Throws

None.

Additional Information

There is an equivalent function, **get-logical-name**, for use in the external Monk environment with Generic Monk e*Ways.

displayb

Syntax

```
(displayb string)
```

Description

Displays the specified string in both literal and hexadecimal formats.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

Unspecified.

Example

```
(displayb "Hello, world\n")
=> 48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 0A |  Hello, world
```

encrypt-password

Syntax

```
(encrypt-password username password)
```

Description

Creates an encrypted password, using the specified username as a key.

Parameters

Name	Type	Description
username	string	The user name
password	string	The password (in clear)

Return Value

String
Returns the encrypted password.

Example

```
(encrypt-password "Administrator" "mypwd")   =>523AA853EFF
```

event-send

Syntax

```
(event-send alert-category alert-sub-category
info-code custom-code reason-name
event-info-string reason-code
event-detail)
```

Description

Issues a Monitoring Event from any Monk script.

Events can use the standard SeeBeyond event codes, or a “user event” code you can use to communicate status conditions of user-created applications.

Note that *reason-code* is unquoted, since it is an integer rather than a string.

Strings supplied for **event-send** parameters should not contain characters that are used as delimiters in the **EventMsg.ssc** or **NotificationMessage.ssc** structures. Using these characters may cause the events to be incorrectly parsed.

Parameters

Parameter	Type	Possible values	Meaning
alert-category	String	ALERTCAT_STATE_ELEM	Element state
		ALERTCAT_MESSAGE_CONTENT	Message content
		ALERTCAT_STATE_EXTERNAL	External state
		ALERTCAT_OPERATIONAL	Operational
		ALERTCAT_PERFORMANCE	Performance
		ALERTCAT_RESOURCE	Resource
		ALERTCAT_USERDEFINED	User defined
alert-subcategory	String	ALERTSUBCAT_CUSTOM	Custom category
		ALERTSUBCAT_DOWN	Down
		ALERTSUBCAT_UP	Up
		ALERTSUBCAT_UNRESP	Unresponsive
		ALERTSUBCAT_RESP	Responded
		ALERTSUBCAT_CANTCONN	Unable to connect
		ALERTSUBCAT_CONN	Connected
		ALERTSUBCAT_LOSTCONN	Lost Connection
		ALERTSUBCAT_UNUSABLE	Unusable/can’t ID
		ALERTSUBCAT_INTEREST	Content of interest
		ALERTSUBCAT_EXPIRED	Expired
		ALERTSUBCAT_INTHRESH	Input threshold
		ALERTSUBCAT_OUTTHRESH	Output threshold
		ALERTSUBCAT_USERAUTH	User authentication

Parameter	Type	Possible values	Meaning
		ALERTSUBCAT_DELIVERY	Alert delivery
		ALERTSUBCAT_UNQUEUEABLE	Unqueueable
		ALERTSUBCAT_DISKTHRESH	Disk threshold
		ALERTSUBCAT_IQLIMIT	IQ Limit
		ALERTSUBCAT_STATUS	Status
		ALERTSUBCAT_TIMER	Timer
info-code	String	ALERTINFO_NONE	None
		ALERTINFO_FATAL	Fatal
		ALERTINFO_CONTROLLED	Controlled
		ALERTINFO_USER	User
		ALERTINFO_LOW	Low
		ALERTINFO_HIGH	High
		ALERTINFO_IOFAILED	IO Failure
		ALERTINFO_BELOW	Below
		ALERTINFO_ABOVE	Above
custom-code	String	any one-byte (printable) character	Any meaning required for user application
reason-name	String	descriptive string	Reason that the event (described by reason-code) occurred
event-info-string	String	Reserved for user agents or other applications using SeeBeyond's API to create Monitoring Events that use this field	Example gives, "This is a bad message"
reason-code	integer	Status or error code	Status/error code sent by the operating system or by the application generating the event
event-detail	list of lists	Reserved for future use. In this field, always enter just the (list) command, which will generate an empty list	

Return Value

integer

Returns 0 if successful. Otherwise, it returns -1.

Examples

```
(event-send "ALERTCAT_MESSAGE_CONTENT" "ALERTSUBCAT_UNUSABLE"
```

```
"ALERTINFO_NONE" "0" "Bad Sequence" "This is a bad message" 0 (list))
    =>  -1
```

Note: *This function is not compatible with* ***stctrans.exe*** *or with the Monk Test Console.*

file-set-creation-mask

Syntax

```
(file-set-creation-mask protectionValue)
```

Description

file-set-creation-mask sets the default permission for new files (similar to UNIX **unmask**).

Parameters

Name	Type	Description
protectionValue	integer	A five-digit integer representing the file creation mask. The first two digits from the left must be zero. The remaining digits represent the protections assigned to owner, group, and world in that order.

Return Value

string

Returns an empty string.

Examples

```
(file-set-creation-mask 00700)        sets default protection to
00700
```

Sets the protection to owner: read, write, execute, all others no access

```
(file-set-creation-mask 000755)        sets default protection to
0075
```

Sets the protection to owner: read, write, execute, all others read, execute, no write.

Additional Notes

The protection system uses the following values:

Protection value	Meaning
00700	read, write, execute: owner (No access by group or other)
00400	read permission: owner (No access by group or other)
00200	write permission: owner (No access by group or other)
00100	execute permission: owner (No access by group or other)
00070	read, write, execute: group
00040	read permission: group (No write or execute permissions)
00020	write permission: group

Protection value	Meaning
00010	executed permission: group
00007	read, write, execute permission: other
00004	read permission: other
00002	write permission: other
0001	execute permission: other
00755	read, write, execute permission: owner write, execute permission: group write, execute permission: other

get-data-dir

Syntax

```
(get-data-dir)
```

Description

Returns the value of the **SystemData** parameter in the **.egate.store** file.

Parameters

None.

Return Value

string
Returns the value of the **SystemData** parameter in the **.egate.store** file.

Example

```
(get-data-dir)
    =>  d:\eGate\client
```

Note: *This function is not compatible with* ***stctrans.exe*** *or with the Monk Test Console.*

reg-retrieve-file

Syntax

```
(reg-retrieve-file file registry_path)
```

Description

Retrieves a file from the e*Gate Registry.

If a file of the same name already exists in the local file system, **reg-retrieve-file** will only overwrite the file if the local file has changed. The function makes this determination by comparing a hash of the local file to a cached hash of the file in the Registry. See the entry for **stcregutil.exe** in the *e*Gate Integrator System Administration and Operations Guide* for more information.

Parameters

Name	Type	Description
file	string	The name of the file to be retrieved
registry_path		The path to the file within the e*Gate Registry

Return Value

string

Returns the pathname to the downloaded file on the local file system if the file exists. Otherwise, returns the name of the non-existent requested file.

Example

```
(reg-retrieve-file "Notification.tsc" "/monk_scripts/common")
=>d:\eGate\client\monk_scripts\common\Notification.tsc
```

Note: *This function is not compatible with* ***stctrans.exe*** *or with the Monk Test Console.*

22.4 Monk Utility Functions

The Monk Utility functions are contained in the **stc_monkutils.dll** file. To use these functions, you must use the **load-extension** function to load the Monk extension file **/eGate/client/bin/stc_monkutils.dll**.

The Monk Utility Functions include:

ascii->ebcdic

Syntax

```
(ascii->ebcdic input [fill-length fill-char] [:Full])
```

Description

Converts an ASCII character or string to an EBCDIC character or string, using a one-for-one lookup table. For example, a “B” character on an ASCII machine (hex 42) is converted to a “B” character on an EBCDIC machine (hex c2).

Optionally, fills the end of the output string with a *fill-char*. The *fill-char* chosen will also be converted to the corresponding EBCDIC character. The parameters, *fill-length* and *fill-char* are used as an optional pair. (Although optional, one is not used without the other.)

The keyword, *:Full*, enables full conversion of both printable and not-printable characters, while the default converts non-printable characters to NULL. The keyword *:Full* parameter must appear as the last parameter. This option uses IBM-1047 for EBCDIC, and ISO-850 for ASCII.

Parameters

Name	Type	Description
input	string	ASCII string to convert.
fill-length	integer	Number of characters to fill.
fill-char	char	Pad character.
:Full	keyword	The use of the keyword enables full conversion of non-printable and printable characters.

Return Value

character
If a character is input, returns an EBCDIC character corresponding to the ASCII version of the original character.

string
If a string is input, returns a string of EBCDIC characters corresponding to the ASCII version of the original string.

Note: *EBCDIC characters displayed on an ASCII machine display differently from the same characters displayed on an EBCDIC machine. The character displayed is the ASCII version of the underlying hex representation of the EBCDIC character. For example, and EBCDIC “â” (hex 42) displays as “B” on an ASCII machine.*

Examples

These examples were created on an ASCII machine.

```
(ascii->ebcdic #\&)                 => P

(ascii->ebcdic #\+)                 => N
```

```
(ascii->ebcdic "cat")                  => âüú

(ascii->ebcdic "cat" 15 #\x67)         => âüúçççççççççççç
```

This example creates a file containing a list of all the conversions:

```
(define a2e_out (open-output-file "a2e-output.txt"))
(load-extension "stc_monkutils.dll")
(do ((i 0 (+ i 1)))
    ((= i 256))
    (begin
        (display (string-append (number->string i) " = ") a2e_out)
        (write-exp (ascii->ebcdic (integer->char i)) a2e_out)
        (newline a2e_out)
    )
)
```

Notes

By default, Monk converts all alphanumeric characters plus the following subset of the ASCII character set.

- alert
- bell
- space
- newline
- formfeed
- carriage return
- horizontal tab
- vertical tab
- backspace
- octal 3-digits
- hexadecimal 2-digits
- a-z, A-Z
- 0 - 9

The ASCII codes that are not translated are:

- 0x00–0x03
- 0x0b–0x13
- 0x18–0x19
- 0x1c–0x1f

The ASCII codes that do not have an EBCDIC equivalent and are translated into an arbitrary EBCDIC code are:

- ASCII 0x5b '[' = EBCDIC 0x4a 'cent character'
- ASCII 0x5d ']' = EBCDIC 0x5a '!'
- ASCII 0x5e '^' = EBCDIC 0x5f 'top-right character'

The ASCII code that does not have an EBCDIC equivalent but is translated into a non-equivalent character is:

- ASCII 0x21 '!' = EBCDIC 0x4f '|'

base64->raw

Syntax

```
(base64->raw string)
```

Description

Converts a base-64 string to a character string.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
Returns a string.

Example

```
(base64->raw "SGVsbG8gd29ybGQ=")      => Hello world
```

binary->string

Syntax

```
(binary->string string)
```

Description

Converts a binary string into a string representation of a number.

The binary string used as input must be in the "big-endian" format.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
Returns a string.

Example

```
(binary->string (string->binary "12345" 3))        =>12345
```

change-directory

Syntax

```
(change-directory string)
```

Description

Changes the working directory of the current process to the specified directory.

Parameters

Name	Type	Description
string	string	A directory name.

Return Value

Boolean
Returns **#t** if the function executes successfully; otherwise returns **#f**.

Example

```
(change-directory "monk_scripts/common/myscripts")    =>#t
```

close-pipe

Syntax

```
(close-pipe handle)
```

Description

Closes the file handle created by the **open-pipe** function. For more information, see **open-pipe** on page 610.

Note: *This command is only available under the UNIX operating system.*

Parameters

Name	Type	Description
handle	string	The name of the file handle to be closed.

Return Value

boolean
Returns a **#t** if the handle is valid. Otherwise, returns **#f**.

Example

```
(define fp (open-pipe "/bin/ls -la"))
(define data "")
(do ((done 0 (+ done 0))) ((= done 1))
  (set! data (read-line fp 1024))
  (if (eof-object? data)
    (begin
      (set! done 1)
    )
    (begin
      (display data)(newline)      => output of ls -la command
    )
  )
)
(close-pipe fp)
```

ebcdic->ascii

Syntax

```
(ebcdic->ascii input [fill-length fill-char] [:Full])
```

Description

Converts an EBCDIC character or string into an ASCII character or string, using a one-for-one lookup table. For example, a “B” character on an EBCDIC machine (hex c2) is converted to a “B” character on an ASCII machine (hex 42).

Optionally, fills the end of the output string with a *fill-char*. The *fill-char* chosen will also be converted to the corresponding EBCDIC character. The parameters, *fill-length* and *fill-char* are used as an optional pair. (Although optional, one is not used without the other.)

The keyword, *:Full*, enables full conversion of both printable and not-printable characters, while the default converts non-printable characters to NULL. The keyword *:Full* parameter must appear as the last parameter. This option uses IBM-1047 for EBCDIC, and ISO-850 for ASCII.

Parameters

Name	Type	Description
input	string	EBCDIC string to convert.
fill-length	integer	Number of characters to fill.
fill-char	char	Pad character.
:Full	keyword	The use of the keyword enables full conversion of non-printable and printable characters.

Return Value

character
If the input is a character, returns the ASCII version of the EBCDIC character.

string
If the input is a string, returns an ASCII string.

Note: *ASCII characters displayed on an EBCDIC machine display differently from the same characters displayed on an ASCII machine. The character displayed is the EBCDIC version of the underlying hex representation of the ASCII character. For example, an ASCII “â” (hex e2) displays as “S” on an EBCDIC machine.*

Examples

These examples were created on an ASCII machine.

The first example converts a string. Since this string doesn’t actually originate from an EBCDIC machine, it must be cast as EBCDIC before it can be converted to ASCII.

```
(define ebcdic_str "")
;Use the string-type! function to cast the string as EBCDIC.
(set! ebcdic_str (string-type! :EBCDIC(ascii->ebcdic "cat")))
```

```
(display ebcdic_str)                    => âüú
(newline)
(display (ebcdic->ascii ebcdic_str))   => cat

(ebcdic->ascii #\x50)                   => &

(ebcdic->ascii #\x6d)                   => _
```

This example creates a file containing a list of all the conversions:

```
(define e2a_out (open-output-file "e2a-output.txt"))
(load-extension "stc_monkutils.dll")
(do ((i 0 (+ i 1)))
    ((= i 256))
    (begin
        (display (string-append (number->string i) " = ") e2a_out)
        (write-exp (ebcdic->ascii (integer->char i)) e2a_out)
        (newline e2a_out)
    )
)
```

Notes

By default, Monk converts all alphanumeric characters plus the following subset of the EBCDIC character set.

- alert
- bell
- space
- newline
- formfeed
- carriage return
- horizontal tab
- vertical tab
- backspace
- octal 3-digits
- hexadecimal 2-digits
- a-z, A-Z
- 0 - 9

The following EBCDIC code points are translated according to the IBM 3274 specification:

- EBCDIC 0x4a 'cent' = ASCII 0x5b '['
- EBCDIC 0x4f 'solid |' = ASCII 0x21 '!'
- EBCDIC 0x5a '!' = ASCII 0x5d ']'
- EBCDIC 0x6a '|' = ASCII 0x7c '|'
- EBCDIC 0x5f 'top-right' = ASCII 0x5e '^'

A carriage return may have to be inserted for certain ASCII devices when converting the following:

- EBCDIC 0x15 'nl' = ASCII 0x0a 'lf'

hexdump->string

Syntax

```
(hexdump->string string)
```

Description

Converts a hexdump string (which has been created using **string->hexdump**) to a character string.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
Returns the converted string. If the conversion is not successful, returns an empty string.

Example

```
(hexdump->string "636174")          =>cat
```

See **string->hexdump** on page 620 for more information.

IBMpacdec->string

Syntax

```
(IBMpacdec->string string)
```

Description

Converts an IBM packed decimal to a string.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
The string corresponding to the IBM packed decimal.

Examples

The following examples work on an EBCDIC machine:

```
(IBMpacdec->string (string->IBMpacdec "0x12345C"))    =>12345

(IBMpacdec->string (string->IBMpacdec "0x12345D"))    =>-12345
```

The following example works on an ASCII machine. The inclusion of the identifier **#EBCDIC** indicates to the Monk engine that the string to be converted is in EBCDIC format. Without this identifier, the data would be incorrectly interpreted as ASCII data.

```
(display
    (ebcdic->ascii
        (IBMpacdec->string #EBCDIC"\x01\x23\x4c")
    :Full)
)

=> +01234
```

IBMzoned->string

Syntax

```
(IBMzoned->string string)
```

Description

Converts a IBM zone-decimal string to a string representation of a number.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
The string corresponding to the zone-decimal.

Examples

The following examples work on an EBCDIC machine:

```
(IBMzoned->string (string->IBMzoned "1234E"))     =>12345

(IBMzoned->string (string->IBMzoned "1234D"))     =>-12345
```

The following example works on an ASCII machine. The inclusion of the identifier **#EBCDIC** indicates to the Monk engine that the string to be converted is in EBCDIC format. Without this identifier, the data would be incorrectly interpreted as ASCII data.

```
(display
    (ebcdic->ascii
        (IBMzoned->string #EBCDIC"\xf1\xf1\xf1\xc1")
    :Full)
)

=> +1111
```

open-pipe

Syntax

```
(open-pipe string)
```

Description

Spawns the specified application and returns a file handle from which you can read the application's output.

Note: *This command is only available under the UNIX operating system.*

Parameters

Name	Type	Description
string	string	An executable file or script to be executed.

Return Value

handle
Returns a file handle.

Example

```
(define fp (open-pipe "/bin/ls -la"))
(define data "")
(do ((done 0 (+ done 0))) ((= done 1))
  (set! data (read-line fp 1024))
  (if (eof-object? data)
    (begin
      (set! done 1)
    )
    (begin
      (display data)(newline)      => output of ls -la command
    )
  )
)
(close-pipe fp)
```

pacdec->string

Syntax

```
(pacdec->string string digit_after)
```

Description

Converts a packed decimal string to a string representation of a number.

Parameters

Name	Type	Description
string	string	The string to be converted
digit_after	integer	The number of digits after the decimal point

Return Value

string
Returns a quoted number (a string).

Example

```
(define mypacdec (string->pacdec "123.12345" 3 5))
(pacdec->string mypacdec 5)          => 123.12345
```

raw->base64

Syntax

```
(raw->base64 string)
```

Description

Converts a raw string into a base-64 string.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
Returns a string.

Example

```
(raw->base64 "Hello, world\n")
```

reg-get-file

Syntax

```
(reg-get-file string)
```

Description

Gets a file from the e*Gate Registry and writes a copy to the default directory.

The file created by this function can be open and read with any of the file access functions.

Parameters

Name	Type	Description
string	string	The string to be converted

Example

```
(reg-get-file "MyDataMap.dat")     => {MONK_UNSPECIFIED}
```

Note: *This function is not compatible with* ***stctrans.exe*** *or with the Monk Test Console.*

sleep

Syntax

```
(sleep time)
```

Description

Waits the specified number of milliseconds, then exits.

Parameters

Name	Type	Description
time	integer	The number of milliseconds to sleep

Return Value

Undefined

Example

```
(sleep 5000)       ; sleep 5 seconds
```

string->7even

Syntax

```
(string->7even string length)
```

Description

Converts a raw string to a string such that for each character, the parity is even and the high bit is set if the count of the remaining seven bits is even.

Parameters

Name	Type	Description
string	string	The string to be converted
length	integer	The length of the string

Return Value

string
Returns a string.

Example

```
(string->7even "ABCDEFG" 7)
```

string->8none

Syntax

```
(string->8none string length)
```

Description

Resets the high-order bit of each character within a string.

This function is the complement of **string->7even**.

Parameters

Name	Type	Description
string	string	The string to be converted
length	integer	The length of the string

Return Value

string
Returns a string.

Example

```
(define mystring (string->7even "ABCDEFG" 7))
(string->8none mystring)          =>ABCDEFG
```

string->binary

Syntax

```
(string->binary string bytes)
```

Description

Converts a string representation of an integer to a blob representation of a big-endian number.

Parameters

Name	Type	Description
string	string	The string to be converted
bytes	integer	The number of bytes in the resulting string. Valid values are 1, 2, 3 or 4.

Return Value

string
Returns a string.

Example

```
(binary->string (string->binary "12345" 3))    =>12345
```

string-decrypt

Syntax

```
(string-decrypt key string)
```

Description

Decrypts the specified string using the specified key.

Parameters

Name	Type	Description
key	string	The encryption key
string	string	The string to be decrypted

Return Value

string
Returns a string.

Example

```
(string-decrypt "key" "06C22BA54DC811")    => mypass
```

string-encrypt

Syntax

```
(string-encrypt key string)
```

Description

Encrypts the specified string using the specified key.

Parameters

Name	Type	Description
key	string	The encryption key
string	string	The string to be encrypted

Return Value

string
Returns a string.

Example

```
(string-encrypt "key" "mypass")      => 06C22BA54DC811
```

See also **encrypt-password** on page 590.

string->hexdump

Syntax

```
(string->hexdump string)
```

Description

Converts a character string to a hexdump string.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
Returns a string.

Example

```
(string->hexdump "cat")       =>636174
```

See **hexdump->string** on page 607 for more information.

string->IBMpacdec

Syntax

```
(string->IBMpacdec string)
```

Description

Converts a string to an IBM packed decimal number.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string

Example

```
(string->IBMpacdec (IBMpacdec->string "12345"))     =>0x12345C

(string->IBMpacdec (IBMpacdec->string "-12345"))    =>0x12345D
```

In the above examples, the output is equal to 3-bytes, and the alpha character represents the sign.

string->IBMzoned

Syntax

```
(string->IBMzoned string)
```

Description

Converts a string to an IBM zone-decimal.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string

Examples

The following examples work on an EBCDIC machine. E is equivalent to xC5, or a positive sign. The N is equivalent to xD5, or a negative sign.:

```
(string->IBMzoned (IBMzoned->string "12345"))    =>1234E

(string->IBMzoned (IBMzoned->string "-12345"))    =>1234N
```

In the above examples, the E is equivalent to xC5, or a positive sign. The N is equivalent to xD5, or a negative sign.

The following example works on an ASCII machine. The inclusion of the identifier **#EBCDIC** indicates to the Monk engine that the string to be converted is in EBCDIC format. Without this identifier, the data would be incorrectly interpreted as ASCII data.

```
(display
    (string->IBMzoned
        (ebcdic->ascii
            (IBMzoned->string #EBCDIC"\xf1\xf1\xf1\xc1") :Full)))

=> 111A
```

string->pacdec

Syntax

```
(string->pacdec string digits_before digits_after)
```

Description

Converts a string representation of a number to a packed decimal string.

Parameters

Name	Type	Description
string	string	The string to be converted
digit_before	integer	The number of digits before the decimal point
digit_after	integer	The number of digits after the decimal point

Return Value

string
Returns a string.

Example

```
(define mypacdec (string->pacdec "123.12345" 3 5))
(pacdec->string mypacdec 5)         => 123.12345
```

string->zoned

Syntax

```
(string-zoned string)
```

Description

Converts a string representation of a number into a zone decimal string.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
Returns a string.

Example

```
(zoned->string (string->zoned "12345"))   =>12345
```

util-util-f-decode

Syntax

```
(util-util-f-decode string)
```

Description

Decodes a url-encoded string.

Parameters

Name	Type	Description
string	string	The url-encoded string to be decoded.

Return Value

string

Returns a url-decoded string if successful. Otherwise, returns **#f** or throws an exception.

Example

```
(util-url-f-decode "12345")   =>12345
```

util-xml-f-decode

Syntax

```
(util-xml-f-decode string)
```

Description

Decodes an xml-encoded string.

Parameters

Name	Type	Description
string	string	The xml-encoded string to be decoded.

Return Value

string
Returns an xml-decoded string if successful. Otherwise, returns **#f** or throws an exception.

Example

```
(util-xml-f-decode "12345")    =>12345
```

zoned->string

Syntax

```
(zoned->string string)
```

Description

Converts a zone-decimal string to a string representation of a number.

Parameters

Name	Type	Description
string	string	The string to be converted

Return Value

string
Returns a string.

Example

```
(zoned->string (string->zoned "12345"))    =>12345
```

Chapter 23

Exception Functionality

The exception functions include:

23.1 Try-Throw-Catch Basics

The try-throw-catch exception and handling mechanism enables the Monk environment to automatically generate exceptions for detected error conditions. You can trap and execute exception handlers for most of these errors. This book includes a list of internally-generated exceptions which can be trapped, along with the standard Monk Exception Codes. You can also define your own exceptions, and cause them to be thrown as required.

The code fragment below shows a simple implementation of the exception handling interface.

```
(display "Starting the test.") (newline)

(define-exception e555 3)
(define (display-exception-info)
  (newline)
  (display (string-append "Exception category: "
           (number->string (exception-category)) "."))
  (newline)
  (display (string-append "Exception symbol: "
           (symbol->string (exception-symbol)) "."))
  (newline)
  (display (string-append "Exception string: " (exception-string)
"."))
  (newline))
```

```
(try
 (display "In Level 1 of try structure.") (newline)
 (throw e555 "My exception")
 (catch
  ((e555) (display "In Level 1 exception code.")
          (display-exception-info))
  (otherwise (display "In Level 1 otherwise stanza.")
         (display-exception-info))
  )
 )
```

The above example defines an exception handler using the **define-exception** Monk routine. The routine accepts two parameters: one is the name (actually a symbol) representing the exception, and the other is the exception category. The symbol should be set to a unique value. The category can be used to group exceptions for later processing purposes. This definition must occur outside of the code block wherein the exception is to be trapped.

You encapsulate the code to trap exceptions within a **try** block, which has the form:

```
(try
 ... main body of code ...
 (catch
  ((exception-symbol to catch)
    ... exception handling code ...
  )
  (otherwise
    ... exception handling code ...
  )
  (always
    ... exception handling code ...
  )
 )
)
```

Within the main code body, you can throw exceptions, or the system may detect an error and throw an exception. When an exception occurs, processing control is immediately passed onto the **catch** stanza within the **try** block, which then attempts to handle the exception. There are three possible entries within the **catch** block:

- specific symbols for exceptions,
- the keyword “otherwise”, which is executed if the symbol of the exception is not explicitly included in the catch list, and
- the keyword “always”, which is always executed if the stanza exists.

All of these entries are optional.

Three additional monk functions are available to support processing of exceptions. These are:

- **exception-category** - returns the category of the current exception,
- **exception-symbol** - which returns the symbol of the current exception, and
- **exception-string** - which returns an error string, including the string which was included when the exception was defined to the system.

If a specific case is present for the thrown exception, the associated code is processed and the system marks the exception as handled. If an “always” stanza exists, it is then processed and processing continues with the next valid code after the end of the **try** block.

If the specific case is not present for the thrown exception, but the "otherwise" stanza exists, the **catch** block executes the code associated with the "otherwise" stanza. If an "always" stanza exists, it is then processed and processing continues with the next valid code after the end of the **try** block.

If the specific case is not present for the thrown exception, and the "otherwise" stanza does not exist, the **catch** executes the "always" clause, if it exists. The exception is not marked as handled, but is passed out of the **try** block. If the block is at the top level, the exception causes the system to return an error. If the **try** block is encapsulated within another **try**, the exception is immediately passed to the **catch** block within the encapsulating **try** block, and exception processing continues as described above.

Note: *When using user-defined exceptions and the define-exception function, you must first check to see if the exception has been defined previously. Your collaboration rule will fail if you attempt to define, for a second time, the user-defined exception. This most likely will result when an e*Way is always running and a new data file becomes available for processing. In this case, the .tsc is executed a second time and the define-exception statement also is executed, unless it is part of an IF statement checking to see if already defined.*

23.1.1 e*Gate Events and Monk Exceptions

If a Monk exception occurs while a BOB or an e*Way is processing an Event, no data is lost. e*Gate protects the data by doing one of the following:

- If the exception occurs while processing an Event within the e*Gate system (specifically, an Event that had been published to an IQ and to which another e*Gate component was subscribing), the Event will be rolled back, and will remain in the IQ.
- If the exception occurs while processing an Event that had been received from the external system, the e*Way will NAK the external system.

abort

Syntax

```
(abort message)
```

Description

Generates an exception in which the *message* will become part of the exception explanation.

Parameters

Name	Type	Description
message	string	The message to display. Optional.

Return Value

Creates an exception condition similar to **throw**.

Examples

```
(abort "Aborting function")
                         => abort:Aborting function
(abort)                  => abort:
```

catch

Syntax

```
(catch
    [ ((exception ...)  expression ...)]
    [[ ((exception ...)  expression ...)]]
    ....
    [ (otherwise  expression ...)]
    [ (always  expression ...)]
)
```

Description

Indicates which exceptions are to be processed and provides the code for processing.

You may have more than one list of exceptions with their associate expressions.

The **catch** must be used within the context of the **try** block. If not within the **try** block, it is ignored.

The following exception types ***are not*** catchable:

- Exception-None
- Exception-Catastrophic

The following exception types *are* catchable:

- Exception-Generic
- Exception-Verify
- Exception-NotVerify
- Exception-FileLookup
- Exception-Mapping
- Exception-CallArgUsage
- Exception-PathInvalid
- Exception-Interface
- Exception-InvalidArg
- Exception-Domain
- Exception-Range
- Exception-Monk-Usage
- Exception-Abort
- Exception-Regex-Failure
- Exception-File
- Exception-System

The following exception type is catchable outside **'(load ..)'** but ignored in **'(load-directory . . .)'**:

- Exception-Parser

The following exception type is not registered with the system:

- Exception-Unknown

Refer to **Exception Codes** on page 641 for a complete listing of all exception codes.

Parameters

None.

Return Value

The **catch** is not entered unless there is an active exception. If the current active exception matches one of the listed exceptions to be caught or the otherwise clause, then the return value is the result of evaluating the last expression. If the always clause exists, the expressions that follow are evaluated and the exception remains active, unless a new one is generated. If the exception is caught, the result is the result of the clause that catches the expression. If the exception is not caught (this includes always), there is no result and the exception is not terminated.

Example

Refer to the **try** example.

define-exception

Syntax

```
(define-exception exception category)
```

Description

Defines an exception category in addition to exception categories pre-defined by the system.

Exception categories predefined by the have one or many individual error messages associated with it. User defined exception categories are not associated with individual error messages. They are used to all user programs to participate in the (try .. (catch...)) functionality.

Parameters

Name	Type	Description
exception	symbol	The symbol that represents the exception.
category	integer	Must be greater than zero.

Return Value

Unspecified.

Example

```
(define-exception e555 3)
```

exception-category

Syntax

```
(exception-category)
```

Description

Retrieves the category of the current active exception.

Parameters

None.

Return Value

Returns an integer as follows:

- **zero** - if there is no exception category
- **negative** - if it is a system exception category
- **positive** - if it is a user-defined exception category

Example

```
(exception-cateory) => 3
```

exception-string

Syntax

```
(exception-string)
```

Description

Retrieves the message portion of the current active exception.

Parameters

None.

Return Value

Returns the message included when the exception was generated.

Example

```
Aborting process.
```

exception-string-all

Syntax

```
(exception-string-all)
```

Description

Retrieves the complete string which represents the exception information.

Parameters

None.

Return Value

Returns the entire string representing the exception information.

Example

```
MONKEXCEPT:0194: abort: Aborting process.
```

exception-symbol

Syntax

```
(exception-symbol)
```

Description

Retrieves the symbol of the current active exception.

Parameters

None.

Return Value

Returns a symbol.

Example

```
(exception-symbol) => 555
```

throw

Syntax

```
(throw exception [message])
```

Description

Creates the specified exception condition.

Parameters

Name	Type	Description
exception	symbol	System or user-defined.
message	string	User-defined message.

Return Value

Creates an exception condition.

Example

```
(throw e555 "My exception")
```

Also, refer to the **try** example.

try

Syntax

```
(try expression ... [(catch ...)])
```

Description

Creates a block of code wherein expressions are evaluated sequentially and where errors may be handled when detected.

If an exception (that is, an error) is generated by an included expression, the catch is entered. You may write the catch clause to execute any of several different expressions depending upon the exception that is raised.

There are a number of predefined exceptions for known error conditions. You may also define additional exceptions using the function **define-exception** on page 634.

Parameters

Name	Type	Description
expression	any	May be any expression.

Return Value

The result of evaluating the last expression.

Example

```
(display "Starting the test.") (newline)

(define-exception e555 3)
(define (display-exception-info)
  (newline)
  (display (string-append "Exception category: "
              (number->string (exception-category)) "."))
  (newline)
  (display (string-append "Exception symbol: "
              (symbol->string (exception-symbol)) "."))
  (newline)
  (display (string-append "Exception string: " (exception-string)
"."))
  (newline))

(try
 (display "In Level 1 of try structure.") (newline)
 (throw e555 "My exception")
 (catch
  ((e555) (display "In Level 1 exception code.")
          (display-exception-info))
  (otherwise (display "In Level 1 otherwise stanza.")
         (display-exception-info))
  )
 )
```

Chapter 24

Exception Codes

When an error condition is detected, the system *raises an exception* to indicate its existence. When an exception is raised, it may be detected and handled.

Exceptions fall into categories. When you write a (try ... (catch ...)) block, you will catch one or more *exception categories*. The System exception categories are listed in **Table 7**. The programmer can define additional exception categories using the **define-exception** monk function.

Note: *Exception-None and Exception-Catastrophic may not be caught.*

Table 7 System Exception Categories

Type	Category
Exception-None	0
Exception-Catastrophic	-1
Exception-Generic	-2
Exception-Verify	-3
Exception-NotVerify	-4
Exception-FileLookup	-5
Exception-Mapping	-6
Exception-CallArgUsage	-7
Exception-PathInvalid	-8
Exception-Interface	-9
Exception-InvalidArg	-10
Exception-Domain	-11
Exception-Range	-12
Exception-Monk-Usage	-13
Exception-Abort	-14
Exception-Regex-Failure	-15
Exception-File	-16
Exception-Parser	-17
Exception-System	-18

When an error is detected, an exception code and an exception message are written to the log file of the component in which the error occurred. Table 9 (below) lists all the exception codes that can be generated. In the table, the percent symbol (%) represents a variable that the Monk code inserts into the exception. The “%s:” in front of the exception string is the name of the function generating the exception. The letters following the (%) sign have the meanings shown in Table 8.

Table 8 Error Argument Format Codes

%s	string
%d	decimal number
%ld	long decimal number
%Le	long double (used more for scientific notation)
%Lg	long double
%lu	long unsigned integer
%ul	unsigned integer long
%e	floating point number
%c	character
%u	unsigned
M_PRIi64	platform dependent 64 bit numbers
M_PRIu64	unsigned 64 bit numbers

Table 9 Exception Code Table

Exception Code	Exception String	Category and Description
0000	%s: argument %u must be a sequence.	**Exception-InvalidArg** In function %s, argument %d returned a result that is not a sequence.
0001	%s: arguments (x and y) must NOT satisfy (x == 0 and y<= 0) or (x < 0 and \"y not an integer\").	**Exception-Domain** In function %s, arguments (x and y) must not satisfy (x == 0 and y<= 0) or (x< 0 and \”y not an integer\”.
0002	%s: must have numeric arguments.	**Exception-InvalidArg** A non numeric parameter has been specified in the function %s.
0003	%s: %Lg %d will OVERFLOW.	**Exception-Domain** The mathematical operation of the parameters %e and %e in the function %s will cause an OVERFLOW condition.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0004	%s: %Lg %d will UNDERFLOW.	**Exception-Domain** The mathematical operation of the parameters %e and %e in the function %s will cause an UNDERFLOW condition.
0005	%s: %Lg " M_PRIi64 " will OVERFLOW.	**Exception-Domain** The operation of the parameters %ld and %ld in the function %s will cause an OVERFLOW condition.
0006	%s: %Lg " M_PRIi64 " will UNDERFLOW.	**Exception-Domain** The operation of the parameters %ld and %ld in the function %s will cause an UNDERFLOW condition.
0007	%s: argument %u must satisfy [x > 0].	**Exception-Domain** Argument %d must be within the domain of numbers.
0008	%s: takes numerical arguments.	**Exception-InvalidArg** A non-numeric argument has been specified in the function %s.
0009	%s: argument %u must be a valid path.	**Exception-InvalidArg** In function %s, a parameter, %d, has been specified that is not a valid path name.
0010	%s: argument %u must be a list of strings.	**Exception-InvalidArg** In function %s, a parameter, %d, has been specified that is not a list of string values
0011	%s: takes a number as an argument.	**Exception-InvalidArg** An argument has been specified that is not a number in function %s.
0012	%s: requires %u argument(s).	**Exception-InvalidArg** In function %s, the required number of arguments %d have not been specified.
0013	%s: malloc (%u) failed [strerror (%d)=(%s)].	**Exception-Catastrophic** In function %s, the attempt to allocate memory has failed. The function outputs error event (%s) exception number (%d).
0014	%s: realloc (??, %u) failed [strerror (%d)=(%s)].	**Exception-Catastrophic** In function %s, the attempt to reallocate memory has failed. The function outputs exception event (%s) exception number (%d).

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0015	%s: input arg count (%u) doesn't match formal arg count (%u).	**Exception-InvalidArg** Procedure call, the number of arguments do not match.
0016	%s: fwrite(??) failed [strerror %d) = (%s)].	**Exception-System** The system fwrite(??) call failed.
0017	%s: call of %s () has failed with code %d.	**Exception-Monk-Usage** The init function in load interface has failed.
0018	%s: path message variable \"%s\" is not `type' event.	**Exception-PathInvalid** In function %s, the path variable, %s, entered is not a valid event type.
0019	%s: fprintf(??) failed [strerror(%d)=(%s)].	**Exception-System** In function %s, the fprintf() instruction failed. The function outputs exception event (%s) exception number (%d).
0020	%s: invalid result received from port callback.	**Exception-Monk-Usage** Result of the callback must be the result of making the call (of the original functionality on the port).
0021	%s: invalid result received from C_API call.	**Exception-Monk-Usage** Invalid result received from an API call.
0022	%s: expect last expression to be `catch'.	**Exception-Monk-Usage** An exception was thrown somewhere in try.
0023	%s: strftime(??, %u, %s, ??) failed [strerror(%d)=(%s)].	**Exception-System** In function %s, the strftime() instruction failed. The function outputs exception event (%s) exception number (%d).
0024	%s: argument %u must be an string port.	**Exception-InvalidArg** This in an incorrect argument.
0025	%s: argument %u exceeds valid string length.	**Exception-InvalidArg** This is an internal or system limitation.
0026	%s: %s is not an event structure.	**Exception-InvalidArg** In function %s, the input string is not a valid event structure.
0027	%s: monk stack overflow. Limit is %u.	**Exception-Monk-Usage** Inputting function %s onto the Monk stack caused the stack to overflow. The limit of the stack is %d.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0028	%s: argument %u must be an event.	**Exception-InvalidArg** In function %s, the %d argument is expected to be an event type.
0029	%s: argument %u must be an integer.	**Exception-InvalidArg** In function %s, the %d argument is expected to be an integer.
0030	%s: argument %u must be a char.	**Exception-InvalidArg** In function %s, the %d argument is expected to be a char.
0031	%s: arguments must be strings.	**Exception-InvalidArg** The arguments to function %s must be strings. A non-string argument has been specified.
0032	%s: argument %u is not mutable.	**Exception-Monk-Usage** The function %s attempted to store a value into the location represented by argument %d which is already in use and thus immutable.
0033	%s: %s.	**Exception-Generic** The generic exception event indicator in function %s.
0034	%s:failed.	**Exception-NotVerify** Operation being performed in function %s failed.
0035	%s: string argument %u must be of length > %u.	**Exception-Domain** The length of the string must be %d length.
0036	%s: variable <%s> has not been defined.	**Exception-Monk-Usage** The variable (%s) in function %s has not been defined.
0037	%s: argument %u must be a non-negative number.	**Exception-InvalidArg** In function %s, argument %d has been expressed as a negative number, it must be non-negative.
0038	%s: arguments must be chars.	**Exception-InvalidArg** In function %s, all arguments must be of type character.
0039	%s: argument %u must be a string.	**Exception-InvalidArg** In function %s, argument %u must be of type string.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0040	%s: error opening \"%s\" [strerror(%d)=(%s)].	**Exception-System** In function %s, an exception occurred while trying to open the function. The function outputs exception event (%s) exception number (%d).
0041	%s: invalid syntax.	**Exception-Monk-Usage** The syntax is not correct.
0042	%s: requires %u or more arguments.	**Exception-InvalidArg** Not enough arguments have been entered in function %s. Function requires a minimum of %u arguments.
0043	%s: %s(y=%d,m=%d,d=%d, h=%d,m=%d,s=%d,i=%d) [strerror(%d)=(%s)].	**Exception-System** The date/time format is in exception. The system will output an event (strerror(%d)= (%s)) indicating what element or elements are in error.
0044	%s: argument %u must be a number.	**Exception-InvalidArg** In function %s, argument %u must be of type number.
0045	%s: argument %u is not mutable.	**Exception-Monk-Usage** The function %s attempted to store a value into the location represented by argument %u which is already in use and thus immutable.
0046	%s: argument %u must be an integer and in [0 <= %u , %u].	**Exception-Range** In function %s, the string must be an integer and is in the specified range [%u - %u].
0047	%s: expected arg(s) are %s.	**Exception-Generic** An explanation of the expected arguments.
0048	%s: argument %u <%s> must be a pair.	**Exception-InvalidArg** In function %s, the two arguments %u and %s must be a pair.
0049	%s: bad constant number \"%s\".	**Exception-Parser** The constant number is not valid.
0050	%s: for \"%s\"; required children for serialization not in NofN[%u <= %u <= %u].	**Exception-Range** The minimum number of children are not present for serialization.
0051	%s: argument %u(%u) must be 2, 8, 10, or 16.	**Exception-InvalidArg** In function %s, the argument %u is not one of the required values. The argument must be a 2, 8, 10, or 16.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0052	%s: argument %u must be in domain [-1, 1].	**Exception-Domain** The argument %u being passed in must be within these limits.
0053	%s: delimiter not in data[%u <= %u <= %u].	**Exception-Mapping** An offset has been specified that is not in the data string.
0054	%s: argument %u must be a vector.	**Exception-InvalidArg** In function %s, argument %u must be of type vector.
0055	%s: argument %u must be a positive integer.	**Exception-InvalidArg** In function %s, argument %d must be a positive integer.
0056	%s: unrecognized char constant #\\%s.	**Exception-Parser** The character constant in this file is invalid.
0057	%s: begin delim requires an end delim for node \"%s\".	**Exception-Mapping** A begin delimiter must be parsed with an end delimiter.
0058	%s: argument %u must be a list.	**Exception-InvalidArg** In function %s, argument %u must be a list.
0059	%s: argument %u must be a time.	**Exception-InvalidArg** In function %s, argument %u must be a time.
0060	%s: error closing port [strerror(%d)=(%s)].	**Exception-System** In function %s, an exception occurred while trying to close the port. The function outputs exception event (%s) exception number (%d).
0061	%s: argument %u must be an input port.	**Exception-InvalidArg** In function %s, argument %u is not the required input port number.
0062	%s: argument %u must be an output port.	**Exception-InvalidArg** In function %s, argument %u is not the required output port number.
0063	%s: string \%.*s\ not found in file (%s).	**Exception-FileLookup** In function %s, the expected string \%.*s\ was not found in the file %s.
0064	%s: couldn't find string %s in map.	**Exception-Monk-Usage** The string %s cannot be found in the map.
0065	%s: multiple binding elements for \"%s\".	**Exception-Monk-Usage** Multiple binding forms cannot use the same variable name.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0066	%s: element %u must be symbol.	**Exception-Mapping** In function %s, element %u must be a symbol.
0067	%s: return value mismatch for arg %u. Expected \"%s\".	**Exception-Monk-Usage** In function %s, return value does not match the expected value of \"%s\".
0068	%s: variable must be a symbol, not \"%s\".	**Exception-Monk-Usage** In function %s, the variable must be a symbol, not \"%s\".
0069	%s: %s.	**Exception-Generic** In funxtion %s, argument is invalid.
0070	%s: invalid <Bindings>. Expect ((<variable1> <init1> <step1>) ...).	**Exception-Monk-Usage** The bindings are incorrect.
0071	%s: expected `(test)' expression.	**Exception-Monk-Usage** **do**, **do*** and **condition** require that a some test expression be there.
0072	%s: variable <%s> must be an integer >= 0 or path.	**Exception-PathInvalid** In function %s, a variable <%s> has been specified that is not a string, number, or path.
0073	%s: <%s> evaluates to \"%s\". Not a procedure or interface.	**Exception-Monk-Usage** The function %s was expecting a procedure name. The name specified <%s> evaluates to the name \"%s\" which is not a recognized procedure name.
0074	%s: argument %u must be a proper list.	**Exception-InvalidArg** In function %s, argument %u is not a proper list.
0075	%s: $s.	**Exception-Monk-Usage** In function %s, argument is invalid.
0076	%s: $s.	**Exception-Mapping** In function %s, argument is invalid.
0077	%s: result of `Put' procedure<%s> must be string.	**Exception-Monk-Usage** The result of the `Put' function specified in the map must be a string.
0078	%s: argument %u is not a valid string type.	**Exception-InvalidArg** The function was expecting an argument that resolves to a string.
0079	%s: argument %u is not a valid char type.	**Exception-InvalidArg** The function was expecting an argument that resolves to a character.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0080	%s: argument %u must be a keyword.	**Exception-InvalidArg** The function was expecting an argument that resolves to a keyword.
0081	%s: expected argument %u to be `%s'.	**Exception-Monk-Usage** The function was expecting a keyword.
0082	%s: for path \"%s\", could not convert \"%s\" to a number.	**Exception-PathInvalid** The contents at the specified path could not be converted to a number.
0083	%s: accepts %u or %u arguments.	**Exception-InvalidArg** The number of arguments is incorrect.
0084	%s: unrecognized char token `%s'.	**Exception-Parser** The parser found a token with an invalid character.
0085	%s: file \"%s\"\n\t\tMUST have \".dll\" extension.	**Exception-File** The function %s found a file without the required .dll extension.
0086	%s: argument %u must evaluate to a symbol.	**Exception-InvalidArg** In function %s, argument %u must evaluate to a symbol.
0087	%s: argument %u must be a symbol.	**Exception-InvalidArg** In function %s, argument %u must be a symbol.
0088	%s: first argument of <clause> must be `(<datum1> ...)' or \"else\".	**Exception-Monk-Usage** In function %s, the first argument of the clause is not the expected datum or \"else\".
0089	%s: <clause> must contain at least one <expression> to be evaluated.	**Exception-Monk-Usage** In function %s, a clause is found without at least one expression to be evaluated.
0090	%s: requires at least a <key> and one <clause>.	**Exception-Monk-Usage** Function %s does not contain the required key and at least one clause.
0091	%s: file \"%s\" not readable.	**Exception-File** Function %s can not read the file \"%s\".
0092	%s: argument %u must be a string or symbol.	**Exception-InvalidArg** In function %s, argument %u must be a symbol or string.
0093	%s: path doesn't exist in the event map.	**Exception-PathInvalid** In function %s, the specified path does not exist in the event map.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0094	%s: invalid byte count in `string’ for `char’ conversion.	**Exception-Generic** Cannot convert string because wrong number of bytes.
0095	%s: number not in [%d <= %g < %u].	**Exception-Range** Number is out of range.
0096	%s: attempt to insert zero length path <%s> is invalid.	**Exception-PathInvalid** In function %s, the specified path \"%s\" is bad.
0097	%s: %Le %u will OVERFLOW.	**Exception-Domain** Number is out of range.
0098	%s: %Le %u will UNDERFLOW.	**Exception-Domain** Number is out of range.
0099	%s: invalid result type for consumer \"%s\".	**Exception-Monk-Usage** The function returns the wrong result type for how the result is used.
0100	%s: gettimeofday(??, 0) failed [strerror(%d)=(%s)].	**Exception-System** In function %s, an exception occurred while trying to get the time of day. The function outputs exception event (%s) exception number (%d).
0101	%s: time(??) failed [strerror(%d)=(%s)].	**Exception-System** In function %s, an exception occurred while trying to get the time. The function outputs exception event (%s) exception number (%d).
0102	%s: list does not contain %u elements.	**Exception-Monk-Usage** The list does not contain the specified %ld elements.
0103	%s: trying to divide by zero is a bad idea.	**Exception-Domain** Function %s is trying to divide by zero. This operation causes the system to crash.
0104	%s: empty string not found in file (%s).	**Exception-FileLookup** Function %s can not find the expected empty string in file (%s).
0105	%s: invalid result for consumer \"%s\"; not in range [0 <= %u < %u].	**Exception-Range** Function returns a result to be assigned into a variable that cannot accept it because it is out of range.
0106	%s: argument %u must be a procedure.	**Exception-InvalidArg** In function %s, argument number %u must be a procedure.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0107	%s: argument %u must evaluate to a list.	**Exception-InvalidArg** In function %s, argument %u must evaluate to a list.
0108	%s: invalid use of Keyword \"%s\".	**Exception-Monk-Usage** A Monk keyword is being used outside of a valid context.
0109	%s: expected `symbol' for <variable> in =>\n\t (define <variable> <expression>) | \n\t(define (<variable> <formals>) <body> | \n\t(define (<variable> . <formal>) <body>).	**Exception-Monk-Usage** Failed to identify the lambda form.
0110	%s: number not in [" M_PRIi64 " <= %Lg < " M_PRIu64 "].	**Exception-Range** In function %s, an exception occurred while performing a floating point operation.
0111	%s: invalid use of System Keyword "%s".	**Exception-Monk-Usage** The use of the keyword \"%s\" by function %s is invalid.
0112	%s: calloc(1, %u) failed [strerror (%d)=(%s)].	**Exception-Catastrophic** In function %s, an exception occurred while trying to get the time of day. The function outputs exception message (%s) exception number (%d).
0113	%s: path length for <\?\?\?\?> too long.	**Exception-File** The potential path that is specified exceeds the internal limits.
0114	%s: %s.	**Exception-Parser** Generically prints our errors from the parser.
0115	%s: call to strdup() failed. Probable System Memory Allocation Problem.	**Exception-Catastrophic** In function %s, the attempt to call strdup() function failed. Caused by a probable problem with System Memory allocation.
0116	%s: invalid path \"%s\".\n\tMust have \"~event-name%%pathelement(s)\" or \"%%pathelement(s) \".	**Exception-PathInvalid** An invalid path \"%s\" has been specified in function %s.
0117	%s: must have `string' to place in data tree.	**Exception-Monk-Usage** Data placed into a data tree must be a string.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0118	%s: argument %u must be an integer and in [%u <= %u < %u].	**Exception-InvalidArg** Integer argument is out of range for the function is using it.
0119	%s: invalid <Bindings>. Expect ((<variable1><init1>) ...).	**Exception-Generic** In function %s, the specified bindings are invalid. Function was expecting ((<variable1><init1>) ...).
0120	%s: argument %u must be a list of pairs.	**Exception-InvalidArg** In function %s, the specified argument %u is not an element in a pair.
0121	%s: argument %u must be a list of length %d.	**Exception-InvalidArg** In function %s, the specified argument %d is not part of the list.
0122	%s: path \"%s\" is not valid for this event map.	**Exception-PathInvalid** In function %s, the specified path "%s" does not exist in the Event Type Definition that was applied to this.
0123	%s: element \"%s\" is defined to have a maximum of %u repetitions. An instruction to add repetition %u is an error.	**Exception-PathInvalid** The element "%s" was defined in the Event Type Definition has having a maximum number of repetition. Attempting to exceed this causes this error.
0124	%s: map has more levels(node -> \"%s\") than delimiters.	**Exception-Mapping** The Event Type Definition applied to this Event does not have enough levels for this Event. There must be one delimiter for each level.
0125	%s: %Le " M_PRIu64 " will OVERFLOW.	**Exception-Domain** The parameter will cause an OVERFLOW condition.
0126	%s: %Le " M_PRIu64 " will UNDERFLOW.	**Exception-Domain** The parameter will cause an UNDERFLOW condition.
0127	%s: element \"%s\" is %ld in length and the start byte is %ld.	**Exception-PathInvalid** In function %s, the specified element is invalid. The element has a length of %ld and its starting byte is %ld.
0128	%s: element \"%s\" is %ld in length and the end byte is %ld.	**Exception-PathInvalid** In function %s, the specified element is invalid. The element has a length of %ld and its ending byte is %ld.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0129	%s: path \"%s\" has start (%ld) greater than end (%ld).	**Exception-PathInvalid** In function %s, the start (%ld) of the specified path \"%s\" is greater than its end (%ld).
0130	%s: path \"%s\" is trying to access repetition %u and the map has only defined %u.	**Exception-PathInvalid** The Event Type Definition applied to this Event only defines a certain limit on repetitions for this element, but the function is trying to access a repetition outside that limit.
0131	%s: argument %u must be a path or string.	**Exception-InvalidArg** In function %s: the specified argument %d must be either a path or string.
0132	%s: argument %d must be a string, number, or path.	**Exception-InvalidArg** %s: the specified argument %d must be either a string, a number, or a path.
0133	%s: for argument %u, expected `:keyword <val>’ pairing.	**Exception-InvalidArg** Must be a keyword-value pairing.
0134	%s: argument %d must be a `%s\’.	**Exception-InvalidArg** In function %s, the specified argument in position (%d) is not a required string \"%s\".
0135	%s: may not close standard input, output or error port.	**Exception-Monk-Usage** User may not perform an illegal operation.
0136	%s: argument %d must be an input/output port.	**Exception-InvalidArg** In function %s, the specified argument in position (%d) is not a required input/output pot.
0137	%s: could not convert \"%s\" to number.	**Exception-Monk-Usage** In function %s, could not convert the string \"%s\" to a number.
0138	%s: port is not available.	**Exception-Monk-Usage** In function%s, the specified port is not available.
0139	%s: error closing \"%s\" [strerror(%d)=(%s)].	**Exception-System** In function %s, an exception occurred while trying to close string \"%s\". The function outputs exception event (%s) exception number (%d).

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0140	%s: argument %d(%ld) must be 1, 2, 3 or 4.	**Exception-InvalidArg** In function %s, the argument in position %d must be either 1, 2, 3, or 4.
0141	%s: argument %d must have a length of %d.	**Exception-InvalidArg** In function %s, the argument in position %d must have a length of %d.
0142	%s: path depth position(%d) greater than length(%d).	**Exception-PathInvalid** In function %s, the path depth position (%d) may not be greater than the length(%d).
0143	%s: `array' delimiter required for \"%s\" repetitions.	**Exception-Mapping** Function %s has encountered an invalid array delimiter for the specified Node Array (%s) repetitions.
0144	%s: %s. Notify STC.	**Exception-Catastrophic** Function %s has encountered an exception %s. Notify SeeBeyond.
0145	%s: %Lg %g will OVERFLOW.	**Exception-Domain** The parameters will cause an OVERFLOW condition.
0146	%s: %Lg %g will UNDERFLOW.	**Exception-Domain** The parameters will cause an UNDERFLOW condition.
0147	%s: argument %u must be boolean.	**Exception-Generic** The argument must be boolean.
0148	%s: argument %u must be a list or vector.	**Exception-InvalidArg** In function %s, the argument %d must be a list or vector.
0149	%s: function result must be boolean.	**Exception-Monk-Usage** Function %s returned a value that did not have either a true or false condition.
0150	UNUSED	Not implemented.
0151	%s: resolved template \"%s\" to short.	**Exception-Mapping** In function %s, the resolved template "%s" is too short, it must be eight characters or more.

Table 9 Exception Code Table (Continued)

<table>
<tr><th>Exception Code</th><th>Exception String</th><th>Category and Description</th></tr>
<tr><td>0152</td><td>%s: expected (define-c-api <in-params> <result-type>) \n\t<result-type> => (any|blob|bool|char|double|int|int64|interface|1double|list|symbol|uInt|uInt64|vector|void) \n\t<in-params> => (<result-type>*).</td><td>Exception-Monk-Usage
The (define-c-api) function was not called according to its prototype.</td></tr>
<tr><td>0153</td><td>%s: C_API call param(%u) has `type’ mismatch from definition.</td><td>Exception-CallArgUsage
A parameter in the (define-c-api) function call does not match the definition expected.</td></tr>
<tr><td>0154</td><td>%s: C_API call param(#%u - `%s’) not supported at this time.</td><td>Exception-CallArgUsage
A parameter in the (define-c-api) function call does not match the definition expected.</td></tr>
<tr><td>0155</td><td>%s: must provide repetition number to be counted.</td><td>Exception-PathInvalid
A repetition number to be counted must be provided.</td></tr>
<tr><td>0156</td><td>%s: terminal `>’ missing in path \"%s\".</td><td><b>Exception-PathInvalid</b><br>In function %s, the terminal `>’ is missing in path \"%s\".</td></tr>
<tr><td>0157</td><td>%s: ill-formed delimiter specification \"%s\".</td><td>Exception-Mapping
The delimiter is not correct.</td></tr>
<tr><td>0158</td><td>%s: encountered unresolved delimiter.</td><td>Exception-Mapping
The encountered delimiter is unresolved.</td></tr>
<tr><td>0159</td><td>%s: encoded length not wholly contained in data.</td><td>Exception-Mapping
In function %s, the encoded length is not wholly contained in the data.</td></tr>
<tr><td>0160</td><td>%s: argument %u must be an integer and [0 <= %d < %u].</td><td>Exception-InvalidArg
Argument %d must be an integer within range [0 <= %d < %u].</td></tr>
<tr><td>0161</td><td>%s: for path \"%s\", ByteOffset must be >= 0.</td><td>Exception-PathInvalid
Must have a positive ByteOffset in the path.</td></tr>
<tr><td>0162</td><td>%s: for path \"%s\", EndByte/Length must be >= 0.</td><td>Exception-PathInvalid
Must have a positive EndByte/Length in the path.</td></tr>
<tr><td>0163</td><td>%s: element %d not %s for: \n\t%s.</td><td>Exception-Mapping
A way to create generic events while creating the map and is the only time it’s ever used.</td></tr>
<tr><td>0164</td><td>%s: min rep \"%u\" is larger than max rep \"%u\" for:\n\t%s.</td><td>Exception-Mapping
This is only used while creating the event map.</td></tr>
</table>

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0165	%s: %Le %Le will OVERFLOW.	**Exception-Domain** The mathematical operation of the parameters %Le and %Le in the function %s will cause an OVERFLOW condition.
0166	%s: %Le %Le will UNDERFLOW.	**Exception-Domain** The mathematical operation of the parameters %Le and %Le in the function %s will cause an UNDERFLOW condition.
0167	%s: C_API has not been associated with executable.	**Exception-Monk-Usage** The (define-c-api) function must be associated with an executable but has not been.
0168	%s: invalid use of previously defined Exception \"%s\".	**Exception-Monk-Usage** Cannot redefine a previously defined exception.
0169	%s: exception value must satisfy [0 < %d].	**Exception-Monk-Usage** User-defined exception must be greater than zero.
0170	%s: failed.	**Exception-Verify** Failed.
0171	%s: argument %u must be an exception.	**Exception-InvalidArg** In function %s, argument %d must be an exception.
0172	%s: element `%s' of clause not a defined exception.	**Exception-Monk-Usage** Element `%s' of the clause is not a valid exception.
0173	%s: first element of <clause> must be `(<exception1> ...)' or \"otherwise\" or \"always\".	**Exception-Monk-Usage** The first element is not in a valid form or is not the correct keyword.
0174	%s: expected <%s> to evaluate to a <procedure> following `=>'.	**Exception-Monk-Usage** Evaluation of the parameter is supposed to be a procedure specific to Case statements.
0175	%s: failed to find `symbol_size' arg(%u).	**Exception-CallArgUsage** The function was called without proper symbol_size specified.
0176	%s: failed to find `blob_size' arg(%u).	**Exception-CallArgUsage** The function was called without proper blob-size specified.
0177	%s: invalid element `type' encountered.	**Exception-CallArgUsage** The function was called without an invalid element type.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0178	%s: invalid index(0 <= %u < %u) for vector.	**Exception-CallArgUsage** The function tried to reference a vector element beyond the range of the vector.
0179	%s: invalid `Argument' vector	**Exception-CallArgUsage** The function expected but was not passed a vector argument .
0180	UNUSED	Not implemented.
0181	%s: desired element(%u) `type' does not match arg.	**Exception-CallArgUsage** The function argument did not match expected argument type.
0182	%s: NULL <parameter> passed.	**Exception-CallArgUsage** The function received a NULL argument where a non-NULL argument was expected.
0183	%s: argument %u must be an interface.	**Exception-InvalidArg** In function %s, argument %d is not an interface.
0184	%s: invalid argument(%s) for interface.	**Exception-Interface** Argument(%s) is not valid for the interface.
0185	%s: interface call \"%s\" failed with code(%d).	**Exception-Interface** The function called failed with this code.
0186	%s: interface \"%s\" has no pointer to executable.	**Exception-Interface** Interface \"%s\" has no pointer to the executable.
0187	%s: variable <%s> must resolve to number.	**Exception-PathInvalid** The symbol must resolve to a number.
0188	%s: variable <%s> must resolve to an integer >= 0.	**Exception-PathInvalid** The symbol must resolve to a positive number.
0189	%s: expected <formals> <body>:\n\t <formals> => <var1> ...) | <var> | <var1> ... <varN> . <varN+1>).	**Exception-Monk-Usage** Description must be a valid lambda expression.
0190	%s: in invalid context.	**Exception-Monk-Usage** Must be a comma before the at sign @, not a quasi- quote.
0191	UNUSED	Not implemented.
0192	%s: failed to map event definition to data.	**Exception-Mapping** In function %s, event definition does not agree with data.

Table 9 Exception Code Table (Continued)

Exception Code	Exception String	Category and Description
0193	UNUSED	Not implemented.
0194	%s: %s.	**Exception-Abort** This type of exception is a port abort.
0195	%s: %s.	**Exception-Regex-Failure** This type of exception is a reg-ex failure.
0196	%s: node \"%s\" with `type' ONA/ ANA may not be at leaf.	**Exception-Mapping** Specific nodes may not be leaf, must have at least one child associated with it.
0197	%s: unknown `type' for node \"%s\".	**Exception-Mapping** An unknown type for the node is specified.
0198	%s: invalid byte offset for `type' OF/AF node \"%s\".	**Exception-Mapping** Function %s returned an invalid byte location during mapping.
0199	%s: error %s.	**Exception-File** This is a system failure, function %s can't find open file.
0200	%s: symbol to be created exceeds internal limits.	**Exception-Monk-Usage** The potential symbol contains more than 1,000 characters.

Index

Symbols

D

E

F

N

O

P

Q

R

T

U