Documentation Library

Gamma™ Reference Volume 1

Version 7.2

Cogent Real-Time Systems, Inc.

August 15, 2012

Gamma™ Reference Volume 1: Version 7.2

A dynamically-typed interpreted programming language specifically designed to allow rapid development of control and user interface applications. Gamma has a syntax similar to C and C++, but has a range of built-in features that make it a far better language for developing sophisticated real-time systems.

Published August 15, 2012
Cogent Real-Time Systems, Inc.
162 Guelph Street, Suite 253
Georgetown, Ontario
Canada, L7G 5X7

Toll Free: 1 (888) 628-2028
Tel: 1 (905) 702-7851
Fax: 1 (905) 702-7850

Information Email: info@cogent.ca
Tech Support Email: support@cogent.ca
Web Site: www.cogent.ca

Revision History

Revision 7.2-1 September 2007
Updated DataHub-related functions for 6.4 release of the DataHub.
Revision 6.2-1 February 2005
Simplified TCP connectivity.
Revision 4.1-1 August 2004
Compatible with Cogent DataHub Version 5.0.
Revision 4.0-2 October 2001
New functions in Input/Output, OSAPIs, Date, and Dynamic Loading reference sections.
Revision 4.0-1 September 2001
Source code compatible across QNX 4, QNX 6, and Linux.
Revision 3.2-1 August 2000
Renamed "Gamma", changed function syntax.
Revision 3.0 October 1999
General reorganization and update of Guide and Reference, released in HTML and QNX Helpviewer formats.
Revision 2.1 June 1999
Converted from Word97 to DocBook SGML.
Revision 2.0 June 1997
Initial release of hardcopy documentation.

Copyright, trademark, and software license information.

Copyright Notice

Trademark Notice

END-USER LICENSE AGREEMENT FOR COGENT SOFTWARE

IMPORTANT - READ CAREFULLY: This End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Cogent Real-Time Systems Inc. ("Cogent") of 162 Guelph Street, Suite 253, Georgetown, Ontario, L7G 5X7, Canada (Tel: 905-702-7851, Fax: 905-702-7850), from whom you acquired the Cogent software product(s) ("SOFTWARE PRODUCT" or "SOFTWARE"), either directly from Cogent or through one of Cogent's authorized resellers.
The SOFTWARE PRODUCT includes computer software, any associated media, any printed materials, and any "online" or electronic documentation. By installing, copying or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree with the terms of this EULA, Cogent is unwilling to license the SOFTWARE PRODUCT to you. In such event, you may not use or copy the SOFTWARE PRODUCT, and you should promptly contact Cogent for instructions on return of the unused product(s) for a refund.

SOFTWARE PRODUCT LICENSE

The SOFTWARE PRODUCT is protected by copyright laws and copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE PRODUCT is licensed, not sold.

1. **EVALUATION USE:** This software is distributed as "Free for Evaluation", and with a per-use royalty for Commercial Use, where "Free for Evaluation" means to evaluate Cogent's software and to do exploratory development and "proof of concept" prototyping of software applications, and where "Free for Evaluation" specifically excludes without limitation:
    i. use of the SOFTWARE PRODUCT in a business setting or in support of a business activity,
    ii. development of a system to be used for commercial gain, whether to be sold or to be used within a company, partnership, organization or entity that transacts commercial business,
    iii. the use of the SOFTWARE PRODUCT in a commercial business for any reason other than exploratory development and "proof of concept" prototyping, even if the SOFTWARE PRODUCT is not incorporated into an application or product to be sold,
    iv. the use of the SOFTWARE PRODUCT to enable the use of another application that was developed with the SOFTWARE PRODUCT,
    v. inclusion of the SOFTWARE PRODUCT in a collection of software, whether that collection is sold, given away, or made part of a larger collection.
    vi. inclusion of the SOFTWARE PRODUCT in another product, whether or not that other product is sold, given away, or made part of a larger product.

2. **COMMERCIAL USE:** COMMERCIAL USE is any use that is not specifically defined in this license as EVALUATION USE.

3. **GRANT OF LICENSE:** This EULA covers both COMMERCIAL and EVALUATION USE of the SOFTWARE PRODUCT. Either clause (A) or (B) of this section will apply to you, depending on your actual use of the SOFTWARE PRODUCT. If you have not purchased a license of the SOFTWARE PRODUCT from Cogent or one of Cogent's authorized resellers, then you may not use the product for COMMERCIAL USE.
    A. **GRANT OF LICENSE (EVALUATION USE):** This EULA grants you the following non-exclusive rights when used for EVALUATION purposes:

Software: You may use the SOFTWARE PRODUCT on any number of computers, either stand-alone, or on a network, so long as every use of the SOFTWARE PRODUCT is for EVALUATION USE. You may reproduce the SOFTWARE PRODUCT, but only as reasonably required to install and use it in accordance with this LICENSE or to follow your normal back-up practices.

Subject to the license expressly granted above, you obtain no right, title or interest in or to the SOFTWARE PRODUCT or related documentation, including but not limited to any copyright, patent, trade secret or other proprietary rights therein. All whole or partial copies of the SOFTWARE PRODUCT remain property of Cogent and will be considered part of the SOFTWARE PRODUCT for the purpose of this EULA.

Unless expressly permitted under this EULA or otherwise by Cogent, you will not:

i. use, reproduce, modify, adapt, translate or otherwise transmit the SOFTWARE PRODUCT or related components, in whole or in part;

ii. rent, lease, license, transfer or otherwise provide access to the SOFTWARE PRODUCT or related components;

iii. alter, remove or cover proprietary notices in or on the SOFTWARE PRODUCT, related documentation or storage media;

iv. export the SOFTWARE PRODUCT from the country in which it was provided to you by Cogent or its authorized reseller;

v. use a multi-processor version of the SOFTWARE PRODUCT in a network larger than that for which you have paid the corresponding multi-processor fees;

vi. decompile, disassemble or otherwise attempt or assist others to reverse engineer the SOFTWARE PRODUCT;

vii. circumvent, disable or otherwise render ineffective any demonstration time-outs, locks on functionality or any other restrictions on use in the SOFTWARE PRODUCT;

viii. circumvent, disable or otherwise render ineffective any license verification mechanisms used by the SOFTWARE PRODUCT;

ix. use the SOFTWARE PRODUCT in any application that is intended to create or could, in the event of malfunction or failure, cause personal injury or property damage; or

x. make use of the SOFTWARE PRODUCT for commercial gain, whether directly, indirectly or incidentally.

B. **GRANT OF LICENSE (COMMERCIAL USE):** This EULA grants you the following non-exclusive rights when used for COMMERCIAL purposes:

Software: You may use the SOFTWARE PRODUCT on one computer, or if the SOFTWARE PRODUCT is a multi-processor version - on one node of a network, either: (i) as a development systems for the purpose of creating value-added software applications in accordance with related Cogent documentation; or (ii) as a single run-time copy for use as an integral part of such an application. This includes reproduction and configuration of the SOFTWARE PRODUCT, but only as reasonably required to install and use it in association with your licensed processor or to follow your normal back-up practices.

Storage/Network Use: You may also store or install a copy of the SOFTWARE PRODUCT on one computer to allow your other computers to use the SOFTWARE PRODUCT over an internal network, and distribute the SOFTWARE PRODUCT to your other computers over an internal network. However, you must acquire and dedicate a license for the SOFTWARE PRODUCT for each computer on which the SOFTWARE PRODUCT is used or to which it is distributed. A license for the SOFTWARE PRODUCT may not be shared or used concurrently on different computers.

Subject to the license expressly granted above, you obtain no right, title or interest in or to the SOFTWARE PRODUCT or related documentation, including but not limited to any copyright, patent, trade secret or other proprietary rights therein. All whole or partial copies of the SOFTWARE PRODUCT remain property of Cogent and will be considered part of the SOFTWARE PRODUCT for the purpose of this EULA.

Unless expressly permitted under this EULA or otherwise by Cogent, you will not:

i. use, reproduce, modify, adapt, translate or otherwise transmit the SOFTWARE PRODUCT or related components, in whole or in part;

ii. rent, lease, license, transfer or otherwise provide access to the SOFTWARE PRODUCT or related components;

iii. alter, remove or cover proprietary notices in or on the SOFTWARE PRODUCT, related documentation or storage media;

iv. export the SOFTWARE PRODUCT from the country in which it was provided to you by Cogent or its authorized reseller;

v. use a multi-processor version of the SOFTWARE PRODUCT in a network larger than that for which you have paid the corresponding multi-processor fees;

vi. decompile, disassemble or otherwise attempt or assist others to reverse engineer the SOFTWARE PRODUCT;

vii. circumvent, disable or otherwise render ineffective any demonstration time-outs, locks on functionality or any other restrictions on use in the SOFTWARE PRODUCT;

viii. circumvent, disable or otherwise render ineffective any license verification mechanisms used by the SOFTWARE PRODUCT, or

ix. use the SOFTWARE PRODUCT in any application that is intended to create or could, in the event of malfunction or failure, cause personal injury or property damage.

4. **WARRANTY:** Cogent cannot warrant that the SOFTWARE PRODUCT will function in accordance with related documentation in every combination of hardware platform, software environment and SOFTWARE PRODUCT configuration. You acknowledge that software bugs are likely to be identified when the SOFTWARE PRODUCT is used in your particular application. You therefore accept the responsibility of satisfying yourself that the SOFTWARE PRODUCT is suitable for your intended use. This includes conducting exhaustive testing of your application prior to its initial release and prior to the release of any related hardware or software modifications or enhancements.

   Subject to documentation errors, Cogent warrants to you for a period of ninety (90) days from acceptance of this EULA (as provided above) that the SOFTWARE PRODUCT as delivered by Cogent is capable of performing the functions described in related Cogent user documentation when used on appropriate hardware. Cogent also warrants that any enclosed disk(s) will be free from defects in material and workmanship under normal use for a period of ninety (90) days from acceptance of this EULA. Cogent is not responsible for disk defects that result from accident or abuse. Your sole remedy for any breach of warranty will be either: i) terminate this EULA and receive a refund of any amount paid to Cogent for the SOFTWARE PRODUCT, or ii) to receive a replacement disk.

5. **LIMITATIONS:** Except as expressly warranted above, the SOFTWARE PRODUCT, any related documentation and disks are provided "as is" without other warranties or conditions of any kind, including but not limited to implied warranties of merchantability, fitness for a particular purpose and non-infringement. You assume the entire risk as to the results and performance of the SOFTWARE PRODUCT. Nothing stated in this EULA will imply that the operation of the SOFTWARE PRODUCT will be uninterrupted or error free or that any errors will be corrected. Other written or oral statements by Cogent, its representatives or others do not constitute warranties or conditions of Cogent.

   In no event will Cogent (or its officers, employees, suppliers, distributors, or licensors: collectively "Its Representatives") be liable to you for any indirect, incidental, special or consequential damages whatsoever, including but not limited to loss of revenue, lost or damaged data or other commercial or economic loss, arising out of any breach of this EULA, any use or inability to use the SOFTWARE PRODUCT or any claim made by a third party, even if Cogent (or Its Representatives) have been advised of the possibility of such damage or claim. In no event will the aggregate liability of Cogent (or that of Its Representatives) for any damages or claim, whether in contract, tort or otherwise, exceed the amount paid by you for the SOFTWARE PRODUCT.

   These limitations shall apply whether or not the alleged breach or default is a breach of a fundamental condition or term, or a fundamental breach. Some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, or certain limitations of implied warranties. Therefore the above limitation may not apply to you.

6. **DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS:**

   Separation of Components. The SOFTWARE PRODUCT is licensed as a single product. Its component parts may not be separated for use on more than one computer.

Termination. Without prejudice to any other rights, Cogent may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such an event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.

7. **UPGRADES:** If the SOFTWARE PRODUCT is an upgrade from another product, whether from Cogent or another supplier, you may use or transfer the SOFTWARE PRODUCT only in conjunction with that upgrade product, unless you destroy the upgraded product. If the SOFTWARE PRODUCT is an upgrade of a Cogent product, you now may use that upgraded product only in accordance with this EULA. If the SOFTWARE PRODUCT is an upgrade of a component of a package of software programs which you licensed as a single product, the SOFTWARE PRODUCT may be used and transferred only as part of that single product package and may not be separated for use on more than one computer.
8. **COPYRIGHT:** All title and copyrights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text and ’applets", incorporated into the SOFTWARE PRODUCT), any accompanying printed material, and any copies of the SOFTWARE PRODUCT, are owned by Cogent or its suppliers. You may not copy the printed materials accompanying the SOFTWARE PRODUCT. All rights not specifically granted under this EULA are reserved by Cogent.
9. **PRODUCT SUPPORT:** Cogent has no obligation under this EULA to provide maintenance, support or training.
10. **RESTRICTED RIGHTS:** Use, duplication, or disclosure by the U.S. Government is subject to restrictions as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (OCT 1988), FAR 12.212(a)(1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as appropriate. Manufacturer is Cogent Real-Time Systems Inc. 162 Guelph Street, Suite 253, Georgetown, Ontario, L7G 5X7, Canada.
11. **GOVERNING LAW:** This Software License Agreement is governed by the laws of the Province of Ontario, Canada. You irrevocably attorn to the jurisdiction of the courts of the Province of Ontario and agree to commence any litigation that may arise hereunder in the courts located in the Judicial District of Peel, Province of Ontario.

Table of Contents

1. What is Gamma? 1

2. System Requirements 2

I. Reference ..... ??

I. Symbols and Literals 4

Data Types and Predicates 5

`undefined_p`, `undefined_symbol_p` 8

Literals 9

Predefined Symbols 11

Reserved Words 14

`t` 15

`nil` 16

gamma, **phgamma** 17

II. Operators 19

Operator Precedence and Associativity 20

Arithmetic Operators 21

Assignment Operators 22

Binary Operator Shorthands 23

Bitwise Operators 25

Class Operators 27

Comparison Operators 29

Evaluation Order Operators 30

Increment and Decrement Operators 31

Logical Operators 32

Quote Operators 33

Symbol Character Operators 34

Ternary Operator 35

III. Statements 36

`class` 37

`condition` 39

`for` 40

`function` 41

`if` 43

`local` 45

`method` 47

`progn, prog1` 49

`protect unwind` 50

`switch` 51

`try catch` 53

`while` 55

`with` 56

IV. Core Functions 58

`call` 60

`class_add_cvar` 61

`class_add_ivar` 62

`class_name` 63

`class_of` 64

`defclass` 65

`defmacro, defmacroe` 66

`defun`, `defune`, ... 67

defmethod 68
defvar 69
destroy 70
eq, equal 71
error 73
eval 74
eval_list 75
eval_string 76
force, forceq, forceqq 77
funcall 78
function_args 79
function_body 80
function_name 81
getprop 82
has_cvar 83
has_ivar 84
instance_vars 85
is_class_member 86
ivar_type 87
macro 88
new 90
parent_class 91
print_stack 92
properties 93
quote, backquote 94
require, load 95
set, setq, setqq 97
setprop 98
setprops 99
trap_error 100
unwind_protect 101
whence 102
V. Lists and Arrays 103
append 104
aref 105
array 106
array_to_list 107
aset 108
assoc, assoc_equal 109
bsearch 110
car, cdr, and others 111
cons 112
copy 113
copy_tree 114
delete 115
difference 116
find, find_equal 117
insert 118
intersection 119
length 120
list, listq 121

list_to_array 122
make_array 123
nappend 124
nremove 125
nreplace, nreplace_equal 126
nth_car, nth_cdr 127
remove 128
reverse 129
rplaca, rplacd 130
shorten_array 131
sort 132
union 133
VI. Strings and Buffers 134
bdelete 135
binsert 136
buffer 137
buffer_to_string 138
format 139
make_buffer 141
open_string 142
parse_string 143
raw_memory 145
shell_match 146
shorten_buffer 147
strchr, strrchr 148
strcmp, stricmp 149
string 150
stringc 151
string_file_buffer 152
string_split 153
string_to_buffer 154
strcvt 155
strlen 156
strncmp, strnicmp 157
strrev 158
strstr 159
substr 160
tolower 161
toupper 162
VII. Data Type Conversion 163
bin 164
char 165
char_val 166
dec 167
hex 168
int 169
number 170
oct 171
symbol 172
VIII. Math 173
acos, asin, atan, atan2 174

and, not, or 175
band, bnot, bor, bxor 176
ceil 177
cfand, cfor 178
conf, set_conf 179
cos, sin, tan 180
div 181
exp 182
floor 183
log, log10, logn 184
neg 185
pow 186
random 187
round 188
set_random 189
sqr 190
sqrt 191
Index ..... **??**
Colophon **196**

List of Tables

1. Data Types and Related Predicates ..... ??
1. Integers ..... ??
2. Real numbers ..... ??
3. Special Values ..... ??
4. Strings ..... ??
5. Symbols ..... ??
6. Other Data Types ..... ??
1. Symbols that are predefined in Gamma ..... ??
1. Words reserved in Gamma ..... ??
1. Operator Precedence and Associativity ..... ??

Chapter 1. What is Gamma?

Gamma is an interpreter, a high-level programming language that has been designed and optimized to reduce the time required for building applications. It has support for the Photon GIU in QNX, and the GTK GUI in Linux and QNX 6. It also has extensions that support HTTP and MySQL.

With Gamma a user can quickly implement algorithms that are far harder to express in other languages such as C. Gamma lets the developer take advantage of many time-saving features such as memory management and improved GUI support. These features, coupled with the ability to fully interact with and debug programs as they run, mean that developers can build, test and refine applications in a shorter time frame than when using other development platforms.

Gamma programs are small, fast and reliable. Gamma is easily embedded into today's smart appliances and web devices.

Gamma is an improved and expanded version of our previous Slang Programming Language for QNX and Photon. Gamma is available on QNX 4, QNX 6 and Linux, and is being ported to Microsoft Windows.

The implementation of Gamma is based on a powerful SCADALisp engine. SCADALisp is a dialect of the Lisp programming language which has been optimized for performance and memory usage, and enhanced with a number of internal functions. All references in this manual to Lisp are in fact to the SCADALisp dialect of Lisp.

You could say Gamma's object language is Lisp, just like Assembler is the object language for C. Knowing Lisp is not a requirement for using Gamma, but it can be helpful. All necessary information on Lisp and how it relates to Gamma is in the Input and Output chapter of this guide.

Chapter 2. System Requirements

QNX 6

- QNX 6.1.0 or later.

QNX 4

- QNX 4.23A or later.
- (For Gamma/Photon) Photon 1.14 or later.

Linux

- Linux 2.4 or later.
- (For Gamma/GTK) GTK 1.2.8.
- The SRR IPC kernel module, which includes a synchronous message passing library modeled on the QNX 4 send/receive/reply message-passing API. This module installs automatically, but requires a C compiler for the installation. You can get more information and/or download this module at the Cogent Web Site.

  

  This module may not be necessary for some Gamma applications, but it is required for any use of timers, event handling, or inter-process communication.

I. Reference

Table of Contents

I. Symbols and Literals....4
II. Operators....19
III. Statements....36
IV. Core Functions....58
V. Lists and Arrays....103
VI. Strings and Buffers....134
VII. Data Type Conversion....163
VIII. Math....173

I. Symbols and Literals

Table of Contents

Data Types and Predicates 5
undefined_p, undefined_symbol_p 8
Literals 9
Predefined Symbols 11
Reserved Words 14
t 15
nil 16
gamma, phgamma 17

Data Types and Predicates

— testing for data types.

Unlike many languages, just about every expression in Gamma is a data type. This gives the flexibility to manipulate functions, arrays, lists, classes, instances and so on as if they were data.

The following data types are defined in Gamma. Beside each data type is the name of a function which can be used to test an expression for that type. These functions are called predicates, and will return `t` if the test is true (the expression is that data type), or `nil` if it is false.

Table 1. Data Types and Related Predicates

Type	Predicate	Comments
alist	`alist_p`	An association list. See `assoc`.
array	`array_p`	See `array` and Lists and Arrays.
autotrace	`autotrace_p`	
breakpoint	`breakpoint_p`	
buffer	`buffer_p`	See `buffer`.
builtin	`builtin_p`	
class	`class_p`	See `class`.
cons	`cons_p`	See `cons` and `list`.
constant	`constant_p`	Constants can be assigned or defined. See `defvar` and `::=`.
destroyed instance	`destroyed_p`	See `new (instance)`.
file	`file_p`	See `open` and `open_string`.
fixed-point real	`fixed_point_p`	See Numeric Types.
function	`function_p`	See `function`.
instance	`instance_p`	See `new (instance)`.
integer	`int_p`, `long_p`	See Literals.
list	`list_p`	See `list` and Lists and Arrays.
macro	`macro_p`	See `macro`.
method	`method_p` (obsolete, always returns `nil`)	See `method`.
nil	`nil_p`	See `nil`.
number	`number_p`	Integer and floating point values are both considered numbers. See Literals.
real	`real_p`	See Literals.
registered	`registered_p`	See `register_point`.
string	`string_p`	See Literals and `string`.
sym-alist	`sym_alist_p`	A symbol-indexed association list. See `assoc`.
symbol	`symbol_p`	See Literals.
t	`true_p`	See t.
task descriptor	none	See `locate_task`.

Type	Predicate	Comments
undefined	`undefined_p`	See `undefined_p`.
undefined symbol	`undefined_symbol_p`	See `undefined_symbol_p`.

Predicates

Predicates are used to test a Gamma object for a given type, as listed. If a Gamma object is of that type, the predicate will return the value `t`.

Syntax

```
alist_p (s_exp)
array_p (s_exp)
autotrace_p (s_exp)
breakpoint_p (s_exp)
buffer_p (s_exp)
builtin_p (s_exp)
class_p (s_exp)
cons_p (s_exp)
constant_p (s_exp)
destroyed_p (s_exp)
file_p (s_exp)
fixed_point_p (s_exp)
function_p (s_exp)
instance_p (s_exp)
int_p (s_exp)
list_p (s_exp)
long_p (s_exp)
macro_p (s_exp)
method_p (s_exp)
nil_p (s_exp)
number_p (s_exp)
real_p (s_exp)
registered_p (s_exp)
string_p (s_exp)
sym_alist_p (s_exp)
symbol_p (s_exp)
true_p (s_exp)
none_p (s_exp)
undefined_p (s_exp)
undefined_symbol_p (s_exp)
```

Arguments

any expression

Returns

`t` or `nil`.

Example

Here is an example for the predicate `function_p`. All the other predicates work in a similar way.

```
Gamma> function_p(div);
t
Gamma> function_p(strcmp);
t
Gamma> function_p(5);
nil
Gamma>
```

undefined_p, undefined_symbol_p

undefined_p, undefined_symbol_p — test for undefined types and symbols.

Syntax

```
undefined_p (s_exp)
undefined_symbol_p (s_exp)
```

Arguments

s_exp

Any Gamma or Lisp expression.

Returns

t if the value of *s_exp* is not defined; otherwise nil.

Description

These two functions perform a similar task, checking to see if the *s_exp* is defined. However, they differ in two important ways:

- undefined_p examines the value of *s_exp* directly, whereas undefined_symbol_p expects the value of *s_exp* to be a symbol, and examines the value of that resulting symbol.
- undefined_p evaluates its argument in a protected scope where any "Symbol is undefined" errors will be trapped and disregarded. undefined_symbol_p evaluates its argument without protection, so it is possible that a "Symbol is undefined" error could be thrown if the evaluation of *s_exp* generates such an error.

Example

```
Gamma> a = #xyz
xyz
Gamma> undefined_p (a);
nil
Gamma> undefined_symbol_p (a);
t
Gamma> xyz = t;
t
Gamma> undefined_symbol_p (a);
nil


Gamma> undefined_p (y);
t
Gamma> undefined_symbol_p (y);
Symbol is undefined: y
debug 1>
```

See Also

Data Types and Predicates

Literals

— defined for integers, reals, strings, and symbols.

Integers

An integer is any group of digits defining a number between -2e+31 and 2e+31 - 1. It cannot contain a decimal point or an exponent. Integers have several different literal notations, but regardless of notation, all integers are 32 bit signed numbers. They are flagged internally with their respective notations and Gamma attempts to maintain and return the notation when the integer is printed.

Table 1. Integers

Notation	Description	Example
	Decimal notation	539
0b	Binary notation	0b1011
0o	Octal notation	0o462
0x	Hexadecimal notation	0x35fc
’ ’	Contents are a character.	’M’

Real numbers

A real number is any group of digits defining a number less than -2e+31, greater than 2e+31 - 1, or containing a non-zero mantissa. It can contain a decimal point, and it may end with the letter e followed by a signed exponent.

Table 2. Real numbers

Notation	Description	Example
[0-9].[0-9]e[+|-][0-9]	Double-precision 64 bit floating-point number.	2.56e-7

There are four special floating point values that can be generated by the computer’s floating point processor. These are generated in different ways based on the floating point operation and the operands.

Table 3. Special Values

Notation	Description	Example
`-1.#QNAN`	Quiet Not a Number	See below.
`-1.#INF`	Negative infinity	`-x / 0`
`1.#INF`	Positive infinity	`x / 0`
`-1.#IND`	Indeterminate	`0 / 0` or `sqrt(-1)`

Normally you can only create `-1.#QNAN` by constructing a floating point representation of an illegal bit pattern by casting memory to a float. There is also `NAN`, which is like `QNAN` but will cause the floating point processor to throw an error instead of returning a `NAN` floating point representation.

You can test for an invalid floating point number like this:

```
if (strchr(string(point.value), ’#’) != -1)
    princ ("The point: ", point, " has an invalid floating point value\n");
```

Strings

A string may have any number of characters. The special forms \n, \t, \f and \r denote newline, tab, form feed, and carriage return respectively. The double quote (") and backslash (\) characters may be

embedded in a string by preceding them with a backslash.

Table 4. Strings

Notation	Description	Example
" "	Contents are a string.	"Good morning."

Symbols

Generally, symbol names are made up of alpha-numeric characters and underscores.

Table 5. Symbols

Notation	Description	Example
[a-z,A-Z,0-9]	One or more characters chosen from : a-z, A-Z, 0-9 are valid for symbol names.	Epax15
_	A _ (underscore) is allowed in any part of a symbol name. This symbol is generally used to separate words in a symbol name. The use of this character at the beginning and end of a symbol is reserved for system use.	`my_var_name`
\	Any non-alphanumeric character other than _ must be preceded by a backslash to be used in a symbol name.	Ft\+\$sq

Other Data Types

The literal representation for all other Gamma data types is discussed in the reference entry associated with creating or accessing that data type, as given in the table below.

Table 6. Other Data Types

Data type	Reference entry
Array	`array`
Buffer	`buffer`
List	`list`
Instance	`new`
Function	`function`
Method	`method`
Class	`class`
File	`open`
Task	`locate_task`

Predefined Symbols

— a table.

Table 1. Symbols that are predefined in Gamma

Symbol Name	Description	Accessibility
`_all_tasks_`	The list of tasks opened using `locate_task`.	read-only
`_auto_load_alist_`	A list of rules used by `AutoLoad`.	read/write
`_case_sensitive_`	Used by reader to control case sensitivity. If `nil`, then all symbols are treated as lower-case. Default is `t`.	read/write
`_comma_`	Internal symbol.	not available
`_commasplice_`	Internal symbol.	not available
`_current_input_`	The currently open file for reading.	read-only
`_debug_`	Not used.	not available
`_eof_`	Gamma representation of the end-of-file status from a read operation.	read-only
`_eol_`	Gamma representation of the end-of-line status from a read operation.	read-only
`_error_stack_`	The stack at the time the last error occurred.	read-only
`_eval_silently_`	If set to `t`, then references to undefined symbols are returned as `_undefined_` instead of stopping the program with an error.	read/write
`_eval_stack_`	Contains the definition of the function being currently evaluated.	read-only
`_event_`	The QNX Windows event name.	read-only
`_fixed_point_`	Controls whether calculations with reals are done in double or fixed-point.	read/write
`_gui_`	The name of the graphical user interface that this version of Gamma was compiled against, as a string.	read-only
`_gui_version_`	The version number of the graphical user interface that this version of Gamma was compiled against, as a string.	read-only
`_ipc_file_`	String file used by IPC functions to create buffers for send/receive/reply sequence.	not available
`_jump_stack_`	Internal symbol.	not available
`_last_error_`	String containing last error.	read-only
`_load_extensions_`	List of strings containing shell-match patterns of acceptable input files.	read/write
`_os_`	The name of the operating system (OS) that this version of Gamma was compiled in, as a string.	read-only
`_os_version_`	The version number of the operating system that this version of Gamma was compiled in, as a string.	read-only

Symbol Name	Description	Accessibility
`_os_release_`	The release number of the operating system that this version of Gamma was compiled in, as a string.	read-only
`_require_path_`	List of strings of paths to search for `require` and `require_lisp`.	read-write
`_signal_handlers_`	See `signal`.	read-only
`SIGABRT`	See `signal`.	read-only
`SIGALRM`	See `signal`.	read-only
`SIGBUS`	See `signal`.	read-only
`SIGCHLD`	See `signal`.	read-only
`SIGCONT`	See `signal`.	read-only
`SIGFPE`	See `signal`.	read-only
`SIGHUP`	See `signal`.	read-only
`SIGILL`	See `signal`.	read-only
`SIGINT`	See `signal`.	read-only
`SIGIO`	See `signal`.	read-only
`SIGIOT`	See `signal`.	read-only
`SIGKILL`	See `signal`.	read-only
`SIGPIPE`	See `signal`.	read-only
`SIGPOLL`	See `signal`.	read-only
`SIGPWR`	See `signal`.	read-only
`SIGQUIT`	See `signal`.	read-only
`SIGSEGV`	See `signal`.	read-only
`SIGSTOP`	See `signal`.	read-only
`SIGTERM`	See `signal`.	read-only
`SIGTRAP`	See `signal`.	read-only
`SIGTSTP`	See `signal`.	read-only
`SIGTTIN`	See `signal`.	read-only
`SIGTTOU`	See `signal`.	read-only
`SIGURG`	See `signal`.	read-only
`SIGUSR1`	See `signal`.	read-only
`SIGUSR2`	See `signal`.	read-only
`SIGWINCH`	See `signal`.	read-only

Symbol Name	Description	Accessibility
`_timers_`	An array of active timers, in this format: `[[secs nsecs fires ((s-exp ...)...) number]...]` *secs* The clock time in seconds when the timer was set. *nsecs* The additional nanoseconds when the timer was set. *fires* The set interval of time to fire, in seconds. *s-exp* Action(s) associated with the timer, inside a list of lists. *number* The timer number.	read-only
`_undefined_`	The Gamma representation of the undefined symbol state.	read-only
`_unwind_stack_`	The stack at the time that an error was recovered.	read-only
`&noeval , !`	Symbol directing Gamma to not evaluate the next argument.	not available
`&optional , ?`	Symbol directing Gamma to treat the following argument as optional.	not available
`=>&rest , ...`	Symbol directing Gamma to expect an optional number of arguments starting at last argument. Passed as a list.	not available

Reserved Words

— a table.

The following table provides a list of words which are reserved by the Gamma language. No symbols can be defined by the user that are identical to these reserved words.

Table 1. Words reserved in Gamma

Reserved Word	Used In
`class`	Class declaration
collect	`with` loop
do	`with` loop
else	`if` statement
`for`	`for` loop
`function`	Function declaration
`if`	`if` statement
`local`	Local variable declaration
`method`	Method declaration
tcollect	`with` loop
`while`	`while` loop
`with`	`with` loop

t

`t` — a logically true value.

Syntax

```
t
```

Returns

`t`

Description

The special object, `t`, is a logically true value which has no other meaning. All Gamma objects other than `nil` are logically true, but only the special object `t` is the logical negation of `nil`. `t` is created by a call to not(`nil`), or by reading the symbol `t`.

The predicate `true_p` explicitly tests for the value `t`. However, in all conditional statements, any non-`nil` value is considered to be true for the purpose of the test.

Example

```
Gamma> x = 3;
3
Gamma> x > 2;
t
Gamma> x == 3;
t
Gamma> !nil;
t
Gamma> 10 < 25;
t
Gamma>
```

See Also

, nil

nil

`nil` — the logically false value.

Syntax

```
nil
```

Returns

`nil`

Description

The special value, `nil`, is a zero-length list. It is the only logically false value in Gamma. All other Gamma values are considered to be logically true. A common mistake for first-time Gamma programmers is to treat the number zero as logically false.

Example

```
Gamma> x = 5;
5
Gamma> x > 10;
nil
Gamma> int_p(x);
t
Gamma> real_p(x);
nil
Gamma> !3;
nil
Gamma> !t;
nil
Gamma>
```

See Also

, t

gamma, phgamma

gamma, phgamma — start Gamma and Gamma/Photon from the shell prompt.

Syntax

```
gamma [-options] [program_name [program_arg]...]

phgamma [-options] [program_name [program_arg]...]
```

Options

`-c` *command*

Execute the named command.

`-C`

Declare all constants at startup.

`-d`

Keep file and line # information on all cells .

`-e`

Do not enter interactive mode.

`-f` *filename*

'Require' (read and process) the named file and set the `-e` flag. As many files as desired can be processed by repeating this option. Although the file is run just like the executable named in *program_name*, the two are not the same, because no program arguments can be passed to a file using the `-f` option. When the file has been completely processed, Gamma moves on to the next option, if any, and will not necessarily enter the interactive mode.

`-F`

Declare all functions at startup.

`-G`

Run as Gamma, regardless of name.

`-h`

Print a help message and exit.

`-H` *heapsize*

Set the heap growth rate increment (default 2000).

`-i` *filename*

'Require' the named file. This is identical to the `-f` option, except that Gamma will enter the interactive mode after all options have been processed.

`-I`

Force entry into interactive mode after completion of the named application.

`-L`

Run as Lisp, regardless of name.

`-m`

Do not run the main function automatically.

`-p`

Protect functions from the garbage collector. (Functions should not be redefined.)

`-q`

Do not print copyright notice.

`-s`

Set the local stack size in longwords.

`-V`

Print the version number.

`-X`

Exit immediately (usually used with `-V`).

`program_name`

The name of an executable program.

`program_arg`

The program arguments.

Returns

A Gamma prompt.

Description

This command starts Gamma or Gamma/Photon in interactive mode at the shell prompt. Flags are processed in the order given on the command line, and can appear more than once.

If the name of the executable contains the word 'Lisp', then it will use the Lisp grammar, otherwise it will use the Gamma grammar.

The `-c` and `-f` used together make possible several interesting ways to invoke and use Gamma. For example:

```
gamma -f domainA.g -c "init = methodA(3);" my_application "thing"
```

permits a user to specify a particular file to be processed, perhaps containing application-specific methods, then execute an arbitrary initialization expression, and finally start the intended application with specified arguments.

The `-c` argument used with `-e` has Gamma execute a command and exit without going into interactive mode. For example:

```
gamma -i hanoi.g -c 'princ (hanoi (3), "\n");' -e
```

would load the Tower of Hanoi code, print the solution to the 3-disk hanoi problem, and then exit. (The single quotes are used to hide the double quotes from the shell.)

Example

```
[~/usr/devtools]$ gamma -m
Gamma(TM) Advanced Programming Language
Copyright (C) Cogent Real-Time Systems Inc., 1996-2001. All rights reserved.
Version 4.0 Build 31 at Aug 12 2001 09:57:56
Gamma>
```

II. Operators

Table of Contents

Operator Precedence and Associativity 20
Arithmetic Operators 21
Assignment Operators 22
Binary Operator Shorthands 23
Bitwise Operators 25
Class Operators 27
Comparison Operators 29
Evaluation Order Operators 30
Increment and Decrement Operators 31
Logical Operators 32
Quote Operators 33
Symbol Character Operators 34
Ternary Operator 35

Operator Precedence and Associativity

— a table.

Table 1. Operator Precedence and Associativity

Precedence	Operator	Associativity
Lowest	ELSE	Right
	=	Right
	| |	Left
	& &	Left
	<, >, < =, > =, = =, ! =	Left
	|, &	Left
	-, +	Left
	Unary -, +, !	Left
	^	Left
	+ +, --	Left
	[ ]	Left
	., ..	Left
	( )	Left
Highest	#	Left

The associativity of operators refers to order in which repeated use of the same operator will be evaluated. For example, the expression 1+2+3 will be evaluated as (1+2)+3 since the " + " operator associates the leftmost operator instances first. In contrast, the statement A = B = C will first perform the B = C assignment, and then the result is assigned to A.

Associativity should not be confused with precedence, which determines which one of different operators will be evaluated first. In the example `1+2_3+4`, the multiplication is performed first due to precedence, while the left addition is performed before the rightmost addition due to associativity, causing the expression to be evaluated as `(1+(2_3))+4`.

See Also

Arithmetic Operators, Assignment Operators, Bitwise Operators, Comparison Operators, Increment and Decrement Operators, Logical Operators, Quote Operators

Arithmetic Operators

`Arithmetic Operators` — (+, -, *, /, %)

Syntax

```
number + number
number - number
number * number
number / number
number % number
```

Arguments

number

Any integer or real number. Non-numbers are treated as zero.

Returns

The mathematical result of the operation.

Description

These operators perform simple mathematical operations on their arguments.

+ gives the sum of the two arguments.

- gives the difference between the first and second arguments.

***** gives the product of the two arguments.

/ gives the first argument divided by the second argument.

% gives the modulus of the first argument by the second, that is, the remainder of the integer division of the first argument by the second.

Example

```
Gamma> 5 + 6;
11
Gamma> 12 / 5;
2.3999999999999999112
Gamma> div(12,5);
2
Gamma> 19 % 5;
4
Gamma>
```

Assignment Operators

`Assignment Operators` — (=, :=, ::=)

Syntax

```
symbol = s_exp
symbol := s_exp
symbol ::= s_exp
```

Arguments

symbol

Any valid symbol.

s_exp

Any expression.

Returns

The assigned value.

Description

= is used to assign a value to a variable.

:= is used to assign a value only if no value is currently assigned to the *symbol*. If the *symbol* already has a value then the *symbol* keeps its original value.

::= is used to assign a constant. Once the assignment has been made no changes to the *symbol* are allowed. Attempted changes to the *symbol* will generate an error.

Example

```
Gamma> a = 5;
5
Gamma> a := 6;
5
Gamma> a;
5
Gamma> b ::= 7;
7
Gamma> b = 8;
Assignment to constant symbol: b
debug 1>
Gamma> b ::= 9;
Defvar of defined constant: b
debug 1>
Gamma>
```

See Also

defvar

Binary Operator Shorthands

`Binary Operator Shorthands` — (+=, -=, *=, /=, %=, &=, ^=, <<=, >>=)

Syntax

```
symbol += number
symbol -= number
symbol *= number
symbol /= number
symbol %= number
symbol &= number
symbol ^= number
symbol <<= number
symbol >>= number
```

Arguments

symbol

A symbol with a numeric value.

number

Any integer or real number.

Returns

The value of the *symbol* as operated on with the *number*.

Description

These operators provide a shorthand way of reassigning values to symbols.

+= gives the sum of the *symbol* and the *number*.

-= gives the difference between the *symbol* and the *number*.

***=** gives the product of the *symbol* and the *number*.

/= gives the *symbol* divided by the *number*.

% gives the modulus of the *symbol* by the *number*, that is, the remainder of the integer division of the *symbol* by the *number*.

&= performs the & operation on the *symbol* and the *number*.

^= performs the ^ operation on the *symbol* and the *number*.

<<= performs the << operation on the *symbol* and the *number*.

>>= performs the >> operation on the *symbol* and the *number*.

Example

```
Gamma> a = 5;
5
Gamma> a += 8;
13
Gamma> a;
13
Gamma>
```

See Also

Arithmetic Operators, Assignment Operators, Bitwise Operators

Bitwise Operators

`Bitwise Operators` — (<<, >>, ~ , &, |, ^)

Syntax

```
number << shift;
number >> shift;
~ number
number & number
number | number
number ^ number
```

Arguments

`number`

Any number,

`shift`

The number of bit shifts to perform.

Returns

An integer which is the result of the particular operation.

Description

<<, >> return the first argument with a left or right bitshift operation performed the number of times of the second argument.

~ returns the binary opposite of the *`number`*.

& compares each of the corresponding digits of the two *`numbers`*. If both digits are 1, returns 1 for that place. Otherwise returns 0 for that place.

| compares each of the corresponding digits of the two *`numbers`*. If either those digits is 1, returns 1 for that place. Otherwise returns 0 for that place.

^ compares each of the corresponding digits of the two *`numbers`*. If both digits are the same, returns O for that place. If they are different (ie. 0 and 1) returns 1 for that place.

Examples

```
Gamma> bin(10);
0b1010
Gamma> bin(10 << 1);
0b00010100
Gamma> bin(10 >> 1);
0b0101
Gamma> bin (~10);
0b11111111111111111111111111110101
Gamma> bin(10);
0b1010
Gamma> bin (9);
0b1001
Gamma> bin (9 & 10);
0b1000
Gamma> bin (9 | 10);
0b1011
Gamma> bin (9 ^ 10);
```

```
0b0011
Gamma>
```

See Also

band, bnot, bor, bxor

Class Operators

`Class Operators` — `(.,..)`

Syntax

```
instance.variable = value
instance.variable
instance..variable = value
instance..variable
```

Arguments

`instance`

An instance of a class.

`variable`

An instance variable name.

`value`

A new value to write to the instance variable.

Returns

The value argument.

Description

These operators assign and evaluate object instance values, using familiar C/C++ structure/class reference syntax. The *`instance`* and its instance *`variable`* are separated by a period and the assignment is made using the = assignment operator. Using two periods between *`instance`* and *`variable`* makes the reader interpret the instance variable.

Using either the `.` or the `..` without the = assignment operator causes the *`variable`* to be evaluated at that instance.

Examples

```
Gamma> class cmpny { name; address; }
(defclass cmpny nil [][address name])
Gamma> company = new(cmpny);
{cmpny (address) (name)}
Gamma> company.name = "Acme Widgets";
"Acme Widgets"
Gamma> company.name;
"Acme Widgets"
Gamma> var = symbol("name");
name
Gamma> company..var;
"Acme Widgets"
Gamma>
```

Here is an example of how the `..` syntax can be used to allow an instance of one class to access a method of another class. This can be useful if a parent and child widget have different methods with the same name, and you want an instance of one to use the method of the other.

```
Gamma> class A{}
(defclass A nil [][])
```

```
Gamma> class B{}
(defclass B nil [][])
Gamma> class C B{}
(defclass C B [][])
Gamma> method A.get (){princ("Class A's method.\n");}
(defun A.get (self) (princ "Class A's method.\n"))
Gamma> method B.get (){princ("Class B's method.\n");}
(defun B.get (self) (princ "Class B's method.\n"))
Gamma> a = new(A);
{A }
Gamma> a.get();
Class A's method.
t
Gamma> b = new(B);
{B }
Gamma> b.get();
Class B's method.
t
Gamma> (b..A.get)();
Class A's method.
t
Gamma> (a..B.get)();
Class B's method.
t
Gamma> c = new(C);
{C }
Gamma> (c..A.get)();
Class A's method.
t
Gamma> (c..B.get)();
Class B's method.
t
Gamma>
```

Comparison Operators

`Comparison Operators` — (`!=`, `<`, `<=`, `==`, `>`, `>=`,)

Syntax

```
number != number
number &lt; number
number &lt;= number
number == number
number &gt; number
number &gt;= number
```

Arguments

`number`

Any integer or real number. Non-numbers are treated as zero.

Returns

`!=` `t` if the first *`number`* is not equal to the second, else `nil`.

`<` `t` if the first *`number`* is less than the second, else `nil`.

`<=` `t` if the first *`number`* is less than or equal to the second, else `nil`.

`==` `t` if the first *`number`* Is equal to the second, else `nil`.

`>` `t` if the first *`number`* is greater than the second, else `nil`.

`>=` `t` if the first *`number`* is greater than or equal to the second, else `nil`.

Description

These functions perform a numeric comparison of their arguments. In mathematical (infix) notation, the function would put the first argument on the left side of the comparison, and the second argument on the right side of the comparison.

Example

```
Gamma> 5 < 6;
t
Gamma> 5 > 6;
nil
Gamma> 5.00 == 5;
t
Gamma> "hello" == string("hel","lo");
t
Gamma> a = 5 + 1;
6
Gamma> a;
6
Gamma> a == 5;
nil
Gamma>
```

See Also

`eq`, `equal`, `strcmp`, `stricmp`

Evaluation Order Operators

`Evaluation Order Operators — , ( )`

Syntax

```
symbol , symbol
( symbol operator symbol )
```

Arguments

`symbol`

Any symbol.

`operator`

Any operator.

Determine

Sequence of operation.

Description

Operations before a `,` are performed before those after it.

Operations enclosed by `(` and `)` are performed first.

Examples

```
Gamma> x = 3;
3
Gamma> princ("x = ", x, "\n");
x = 3
t
Gamma> (2 + 3) * 4;
20
Gamma> 2 + (3 * 4);
14
Gamma>
```

Increment and Decrement Operators

`Increment and decrement operators` — (++, --)

Syntax

```
++symbol
symbol++
--symbol
symbol__
```

Arguments

`symbol`

A symbol whose value is a number.

Returns

The value of the symbol plus or minus one.

Description

These operators perform auto-increments or decrements on numeric symbols. When the ++ is placed before a symbol, it performs a pre-increment, where the value is incremented and the result is the symbol's value + 1. When ++ is placed after a symbol, it performs a post-increment. Here the result of the operation is the value of the symbol prior to being incremented. The -- operator works in the same way. These operators only take symbols as arguments. It is not possible to auto-increment or decrement an array element, list element, or instance variable.

Examples

```
Gamma> a = 5;
5
Gamma> ++ a;
6
Gamma> a;
6
Gamma> a ++;
6
Gamma> a;
7
Gamma> a = 5;
5
Gamma> -- a;
4
Gamma> a;
4
Gamma> a --;
4
Gamma> a;
3
Gamma>
```

Logical Operators

`Logical Operators` — (`!`, `&&`, `||`)

Syntax

```
! s_exp
s_exp && s_exp ...
s_exp || !s_exp ...
```

Arguments

s_exp

Any Gamma or Lisp expression.

Returns

Non-`nil` or `nil`.

Description

In Gamma or Lisp, any expression which is not `nil` is treated as being true (`t`) for the purpose of boolean logic. Applying `!` to any non-`nil` expression will produce `nil`. Applying `!` to `nil` must produce an arbitrary non-`nil` result. The generic non-`nil` value in Gamma is `t`.

`&&` evaluates each of its arguments in order, and continues so long as each argument evaluates to non-`nil`. If any argument is `nil`, then `nil` is returned immediately, without evaluating the rest of the arguments. If no argument is `nil`, the last argument is returned.

`||` returns non-`nil` if any of its arguments is not `nil`. Each argument is evaluated in turn, and as soon as a non-`nil` value is reached, that argument is returned. Subsequent arguments are not evaluated.

Examples

```
Gamma> 6;
6
Gamma> !6;
nil
Gamma> !nil;
t
Gamma> 5<6 && string("hi ","there");
"hi there"
Gamma> 5>6 && string("hi ","there");
nil
Gamma> x = 5;
5
Gamma> y = 6;
6
Gamma> (x = t) || (y = 0);
t
Gamma> x;
t
Gamma> y;
6
Gamma>
```

See Also

`and, not, or`

Quote Operators

`Quote Operators` — (#, ‘, @)

Syntax

#s_exp
‘s_exp
@s_exp

Arguments

s_exp

Any Gamma or Lisp expression.

Returns

Does not evaluate the symbol; it returns the protected expression that follows it.

Description

Normally Gamma evaluates every expression as it parses through the code. The # operator protects the contents of an expression from the evaluator. The ‘ operator does the same thing, but allows for evaluation of sub-expressions. Any sub-expression tagged with @ operator that occurs within a back-ticked (‘) expression will be evaluated.

Any error messages involving the @ operator will use the expression _comma_. This is because the @ operator in Gamma corresponds to a comma operator (,) in Lisp syntax. When a Gamma expression is passed to Lisp, the @ operator is converted to a comma. But if the Lisp comma operator is ever read back into Gamma, it is represented by the symbol _comma_ to avoid confusion with the (,) operator used in Gamma function calls.

Examples

```
Gamma> name = "John";
"John"
Gamma> name;
"John"
Gamma> #name;
name

Gamma> x = 4;
4
Gamma> list (1,x);
(1 4)
Gamma> #list (1,x);
(list 1 x)
Gamma> list (1,#x);
(1 x)
Gamma> ‘list (1,x);
(list 1 x)
Gamma> ‘list (1,@x);
(list 1 4)
Gamma>
```

Symbol Character Operators

`Symbol Character Operators` — (\, $)

Syntax

\symbol_character
$symbol_character_string

Arguments

symbol_character

A character that is normally not valid within the string of a symbol name.

symbol_character_string

A symbol name that contains one or more characters that are normally not valid within the string of a symbol name.

Returns

A valid symbol name.

Description

These operators allow you to put non-valid characters into a symbol name. They must be used every time the symbol is written, not just the first time.

\ makes the immediately following character valid.

$ makes the whole string valid, regardless of which individual characters are not normally valid.

Example

```
Gamma>  my\:example1 = 5;
5
Gamma>  x = my\:example1 + 7;
10
Gamma>  $my:example2 = 9;
9
```

Ternary Operator

`Ternary Operator` — `( ? : )`

Syntax

`condition ? s_exp : s_exp`

Arguments

`condition`

Any Gamma or Lisp expression.

`s_exp`

Any Gamma or Lisp expression.

Returns

The first *`s_exp`* if the *`condition`* is true, otherwise the second *`s_exp`*.

Examples

```
Gamma> a = t ? 2 : 8;
2
Gamma> a;
2
Gamma> b = (a == 7) ? 2 : 8;
8
Gamma> b;
8
Gamma>
```

III. Statements

Table of Contents

class 37
condition 39
for 40
function 41
if 43
local 45
method 47
progn, prog1 49
protect unwind 50
switch 51
try catch 53
while 55
with 56

class

`class` — defines a class.

Syntax

```
class class_name [parent]
{
[instance_var [= initial_value];]
...

[static:class_var[= initial_value];]
...

}
```

Arguments

`class_name`

The name of the new class.

`parent`

The parent (base) class.

`class_var`

Class variable definition.

`instance_var`

Instance variable definition. This is provided in the form of a list of variable definitions. Each variable definition is either a variable name or a list which contains a variable name and a default value expression. Whenever a new instance is formed, the default value expression is evaluated to the default value. If no default value is given, the instance variable's value will be `nil`.

`initial_value`

Initial value given to *`instance_var`*, if none then `nil` is assigned to that instance variable.

Returns

A class definition.

Description

This function constructs a class definition and binds the class-name symbol in the current scope to refer to that class. The class mechanism allows only a single parent (base) class. None of the arguments to `class` is evaluated. If *`instance_var`*s are defined with the same names as inherited variables, the inherited variables are overridden and cannot be accessed by instances of this class.

- The `class` statement creates a new class.
  - If the parent (base) class is omitted, or is `nil`, then the resulting class has no parent (base).
  - Each instance variable consists of a name (a symbol) and an optional initial value that will be assigned whenever a new instance of the class is created using the `new` function.
  - The resulting class definition, which is a data object in its own right, will be assigned to the symbol, name.

Example

This example creates two classes: a base class, RegPolygon; and a class derived from it, Square. RegPolygon has two attributes: sides and length. When Square is created, its parent (base) class (RegPolygon) is explicitly assigned. In addition, the attibute sides is assigned a value of 4.

```
Gamma> class RegPolygon{sides; length;}
(defclass RegPolygon nil [][length sides])
Gamma> class Square RegPolygon {sides = 4;}
(defclass Square RegPolygon [][length (sides . 4)])
```

This example creates a class with instance variables and class variables.

```
Gamma> class Other {ivar1; ivar2; static: cvar1; cvar2;}
(defclass Other nil [cvar1 cvar2][ivar1 ivar2])
Gamma>
```

See Also

Class Operators, `class_add_cvar`, `class_add_ivar`, `method`, `new`

condition

`condition` — tests conditions.

Syntax

```
condition {case condition: statement
           [case condition: statement]
            ...
           [default: statement]}
```

Arguments

`condition`

Any Gamma or Lisp expression.

`statement`

Any Gamma statement.

Returns

The return value of the *`statement`* that corresponds to the first true *`condition`* or the default. Otherwise `nil`.

Description

This statement is similar to the `switch` statement, except that it takes no arguments. It checks the truth value of each *`condition`* in turn. The first true *`condition`* encountered returns with the return value of the passed *`statement`*. If no *`condition`* is true, it returns the return value of the default *`statement`*, (or `nil`, if no default statement is given).

The words "case" and "default" and the symbols {, :, and } are unchanging syntactical elements.

Example

```
Gamma> a = 5;
5
Gamma> b = 9;
9
Gamma> condition {case a == b: princ("Equal\n"); case a != b: princ("Unequal\n");}
Unequal
t
Gamma>
```

Also see the example in `switch`.

See Also

`switch`

`for`

`for` — checks a condition and performs a statement.

Syntax

`for (`*`setup ; condition ; iteration`*`) statement`

Arguments

`setup`

An iteration setup, usually a variable with an initial value.

`condition`

The condition to test.

`iteration`

Any Gamma expression, usually used to increment the variable in *`setup.`*

`statement`

Any Gamma statement.

Returns

The value of the *`condition`*.

Description

This statement is essentially identical to a `for` loop in C, and the syntax is the same. It checks a condition iteratively, and executes a statement when the condition is true.

Example

This `for` loop counts from 0 to 10, printing out the value of i as it loops.

```
for (i=0;i<=10;i++)
{
   princ("value of i: ", i, "\n");
}
```

See Also

Statements, `while`, `with`

function

`function` — creates a new function.

Syntax

```
function name ([argument [,argument]... ]) statement
```

Arguments

name

The name of the function.

argument

A symbol that names the argument.

statement

The body of the function.

Returns

A named function definition in Lisp syntax. When a function is called, the return value is the value of the last expression to be evaluated in the function body.

Description

The `function` statement declares a new function. All `function` arguments are implicitly local to the scope of the function. The argument list is denoted by parentheses, and contains zero or more argument definitions, separated by commas. Each *argument* can be a symbol, which is the name of the argument, along with any combination of the following modifiers:

! before the *argument* indicates that this argument will not be evaluated when the function is called.

? after the *argument* indicates that this argument is optional. Only the first optional argument has to be marked as optional. All arguments after that are implicitly optional. An optional argument may have a default value, specified by appending = expression after the question mark.

The only way to test whether an optional function parameter has been provided is by using the predicate `undefined_p`, which tests for the _`undefined`_ value.

... after the *argument* indicates that this argument is a "catch-all" argument used to implement variable length argument lists. Only the last argument in the argument list can have **...** after it. An argument modified by **...** will always be either `nil`, or a list containing all arguments from this position onward in the function call.

When a function is called, its arguments are bound in a new local scope, overriding previous definitions of those symbols for the duration of the function.

Example

This function, with one argument, returns an integer at least one greater than the argument.

```
function next (n)
{
 ceil(n) + 1;
}
```

This function prints its first two arguments and optionally prints a new line (which is printed by default). It returns a string concatenation of the first two arguments.

```
function print_two (first, second, newline?=t)
{
 princ(first, " ", second);
 if (newline)
  terpri();

 string(first, " ", second);
}
```

This function adds all of its arguments. It insists on having at least one argument. Notice that the optional character '?' and the rest character '...' are combined in the second argument.

```
function add_all (first, others...?)
{
 local sum,x = 0;
 sum = first;
 if ( !undefined_p(others) )
 {
  for(x=others;x;x=cdr(x))
   sum = sum + car(x);
 }
 sum;
}
```

See Also

`method`, Statements

if

`if` — conditionally evaluates statements.

Syntax

```
if (condition) statement [else statement]
```

Arguments

`condition`

A Gamma or Lisp expression to test.

`statement`

A statement to perform if the condition is non-`nil`.

Returns

The evaluation of the *`statement`* that corresponds to the satisfied *`condition`*.

Description

The `if` statement evaluates its *`condition`* and tests the result. If the result is non-`nil`, then the *`statement`* is evaluated and the result returned.

The `else` option allows for another statement. If the *`condition`* is `nil`, the `else` statement (if included) is evaluated and the result returned. This statement could be another `if` statement with another condition and `else` statement, etc., permitting multiple `else/if` constructs. The entire `else` option can be omitted if desired.

- If the *`condition`* is `nil` and no else *`statement`* exists, `nil` is returned.
- The `else` part of a nested `if` statement will always be associated with the closest `if` statement when ambiguity exists. Ambiguity can be avoided by explicitly defining code blocks (using curly brackets).

- In interactive mode Gamma has to read two tokens (`if` and `else`) before it will process the statement. If you are not using an `else` part, you have to enter a second semicolon (;) to indicate that the `if` statement is ready for processing.

Example

```
Gamma> x = 5;
5
Gamma> y = 6;
6
Gamma> if (x > y) princ("greater\n"); else princ("not greater\n");
not greater
t
Gamma>
```

The following code:

```
name = "John";
age = 35;

if ((name == "Hank") || (name == "Sue"))
{
    princ("Hi ", name,"\n");
```

```
}
else if ((name == "John") && (age < 20))
{
    princ("Hi ", name," Junior\n");
}
else if ((name == "John") && (age >= 20))
{
    princ("Hi ", name," Senior\n");
}
else
{
    princ("I don't know you\n");
}
```

Will produce the following results:

```
Hi John Senior
```

See Also

Statements, `for`, `while`

local

`local` — allows for implementing local variables within functions.

Syntax

```
local !variable [= s_exp] [, !variable [= s_exp]...];
```

Arguments

`variable`

A symbol.

`s_exp`

Any Gamma or Lisp expression.

Returns

The value of the *`s_exp`*, or `nil` if no value was assigned.

Description

This statement is provided to allow other grammars to implement local variables within functions. It defines new local variables in the current scope, overriding any outer scope that may also define those variables. Each `local` variable consists of a variable name (a symbol) and an optional initial value.

To test whether a symbol is bound in the current scope, use the predicate `undefined_p`.

Example

```
Gamma> i = 5;
5
Gamma> function print_three_local ()
{
 local i;
 for(i=1;i<=3;i++)
 {
  princ("value of i is: ", i, "\n");
 }
}
<function definition>
Gamma> function print_three_global ()
{
// local i;
 for(i=1;i<=3;i++)
 {
  princ("value of i is: ", i, "\n");
 }
}
<function definition>
Gamma> i;
5
Gamma> print_three_local();
value of i is: 1
value of i is: 2
value of i is: 3
3
Gamma> i;
5
```

```
Gamma> print_three_global();
value of i is: 1
value of i is: 2
value of i is: 3
3
Gamma> i;
4
Gamma>
```

This example shows the variable `i` receiving the value of `5`. The two functions are defined identically, except for their names and where the second function comments out the ’local’ command. When the first function is run the internal scoping using the local directive protects the value of `i` globally. When the function returns, `i` remains at `5`, even though it was the value `1`,`2` and `3` within the scope of that function.

The second function has the ’local’ directive commented out (using `//`), so the global variable `i` is modified. When the function returns and we check the value of `i`, it is `4`. The ’for-loop’ within the second function incremented the value of `i` until it failed the `i<=3` comparison. After the second function is run the value of `i` is `4`. The global variable `i` has *not* been protected in the second function.

method

`method` — defines a method for a given class.

Syntax

```
method class.method_name ([argument [, argument]...]) statement
```

Arguments

`class`

The class for which this method is defined.

`method_name`

The name of the method.

`arguments`

The argument list for this method. This does not include the implied argument self, nor is self defined when the arguments are bound as the method is called.

`statement`

The body code statment for this method. Within this statement the special variable `self` is defined as the instance on which this method is being applied.

Returns

A function definition of the resulting method function.

Description

This statement defines a method function for a given class. If a method already exists for this class with this name, the previous definition will be replaced. If a method of the same name exists for any parent (base) class of the given class, it will be overridden for instances of this class only. In Gamma methods are run using the syntax:

```
instance.method_name(arguments);
```

which is the familiar *`object.method`* syntax used in C++.

- The `method` syntax creates a new method for a particular class. It is an error to omit the class.
- The argument list for `method` is identical to the argument list for `function`.

Example

```
Gamma> class RegPolygon{sides; length;}
(defclass RegPolygon nil [][length sides])
Gamma> method RegPolygon.perimeter (){.sides * .length;}
(defun RegPolygon.perimeter (self) (* (@ self sides) (@ self length)))
Gamma> class Square RegPolygon {sides = 4;}
(defclass Square RegPolygon [][length (sides . 4)])
Gamma> method Square.area (){sqr(self.length);}
(defun Square.area (self) (sqr (@ self length)))
Gamma> sqB = new(Square);
{Square (length) (sides . 4)}
Gamma> sqB.length = 3;
3
```

```
Gamma> sqB.perimeter();
12
Gamma> sqB.area();
9
Gamma>
```

See Also

Class Operators, `class`, `defun`, `new`

progn, prog1

`progn, prog1` — group several statements into one expression.

Syntax

```
progn {!statement [!statement] ...}
prog1 {!statement [!statement] ...}
```

Arguments

`statement`

Any valid Gamma statement.

Returns

`progn`: the return value of the last statement.

`prog1`: the return value of the first statement.

Description

These two syntactical elements are not statements, but they transform a group of one or more statements into a single Gamma expression. The value of the resulting expression is the return value of the last statement for `progn` or the first statement for `prog1`. This is useful for performing complex actions where only a single expression is permitted, such as in a callback. No new scope is entered for a `progn` or a `prog1`.

The syntax of these unique elements uses curly braces `{ }`, but don't confuse them with statements. They behave exactly like expressions.

Example

```
Gamma> a = 2;
2
Gamma> b = 5;
5
Gamma> progn{a = 3; princ("a: ",a,"\n"); c = a + b; princ("c: ",c,"\n");};
a: 3
c: 8
t
Gamma> string( prog1{a = 5; b = 1;} );
"5"
Gamma> a;
5
Gamma> b;
1
Gamma>
```

protect unwind

`protect unwind` — evaluates protected code, despite errors.

Syntax

`protect` *`statement`* `unwind` *`statement`*

Arguments

`statement`

Any Gamma or Lisp statement.

Returns

If no error occurs, the result of evaluating the protect *`statement`* and the unwind *`statement`*. If an error occurs, the result of the unwind *`statement`* only.

Description

This function ensures that a piece of code will be evaluated, even if an error occurs within the protect *`statement`* code. This is typically used when an error might occur but cleanup code has to be evaluated even in the event of an error. The error condition will not be cleared by this statement. If an error occurs, control will be passed to the innermost `trap_error` function or to the outer level error handler immediately after the unwind *`statement`* is evaluated.

Example

This code will close its file and run a `write_all_output` function even if an error occurs.

```
if (fp=open("filename","w"))
{
        protect close(fp); unwind write_all_output();
}
```

See Also

`error`, Statements, `try catch`, Tutorial II Error Handling

switch

`switch` — tests arguments with conditions.

Syntax

```
switch (symbol) {case condition:
                      statement
                     [statement...]

                 [case condition:
                       statement
                      [statement...]]
                  ...

                 [default:
                       statement
                      [statement...]]}
```

Arguments

`symbol`

A symbol with a value to test against the value of the *`condition`*(s).

`condition`

Any Gamma or Lisp expression.

`statement`

Any Gamma statement.

Returns

The return value of the *`statement`* that corresponds to the first satisfied *`condition`* or the default. Otherwise `nil`.

Description

This statement is similar to the `condition` statement, except that it takes an argument. It checks the value of the passed *`symbol`* against the value of the *`condition`* for each case in turn. The first match returns the return value of the corresponding *`statement`*. If there is no match, it returns the return value of the default *`statement`*, if any.

The words "case" and "default" and the symbols {, :, and } are unchanging syntactical elements.

Example

```
Gamma> a = "on";
"on"
Gamma> b = 6;
6
Gamma> c = "Nothing";
"Nothing"
Gamma> switch(a) {case "on": 75; case "off": 20; default: 0;}
75
Gamma> switch(b) {case "on": 40; case "off": 10; default: princ("Huh?\n");}
Huh?
t
Gamma> switch(c) {case "on": 1; case "off": 0;}
nil
```

```
    Gamma>

#!/usr/cogent/bin/gamma


/*
  This example demonstrates the switch and condition
  statements.  The switch statement checks the command
  line argument and prints a response.  The case argument
  checks the command line argument and the result of the
  switch statement.
*/

function main ()
{
        a =  number ((cadr(argv)));

        switch (a)
        {
          case 1:
                princ ("One\n");
          case 2:
                princ ("Two\n");
          case 2+1:
                princ ("Three\n");
          case 4:
                princ ("Four\n");
          default:
                princ ("Something else: ", a, "\n");
        }

        condition
        {
          case a == 1:
                princ ("Condition a == 1\n");
          case cadr(argv) == "Hello":
                princ ("Condition a == Hello\n");
          default:
                princ ("No condition met\n");
        }
}
```

See Also

condition

try catch

`try catch` — catches errors in the body code.

Syntax

`try` *`statement`* `catch` *`statement`*

Arguments

`statement`

Any Gamma or Lisp statement.

Returns

If no error occurs, the result of evaluating the `try` *`statement`*. If an error occurs, the result of the `catch` *`statement`*.

Description

This statement catches any errors that may occur while evaluating the `try` *`statement`* code. If no error occurs, then `try catch` will finish without ever evaluating the `catch` *`statement`*. If an error does occur, `try catch` will evaluate the catch *`statement`* code immediately and the error condition will be cleared. This is usually used to protect a running program from a piece of unpredictable code, such as an event handler. If the error is not caught it will be propagated to the top-level error handler, causing the interpreter to go into an interactive debugging session.

Example

The following code:

```
#!/usr/cogent/bin/gamma

try
{
   2 + nil;
}
catch
{
   princ("Error:\n", _error_stack_, "\n");
}
```

Will give these results:

```
Error:
((trap_error #0=(+ 2 nil) (princ Error:
 _error_stack_
)) #0#)
```

The following piece of code will run an event loop and protect against an unpredictable event.

```
while(t)
{
     try (next_event()) catch (print_trapped_error());
}

function print_trapped_error ()
{
     princ("Error:\n", _error_stack_, "\n");
     princ("Clearing error condition and continuing.\n");
}
```

See Also

`error`, Statements, `trap_error`, `protect unwind`, Tutorial II Error Handling

`while`

`while` — iterates, evaluating a statement.

Syntax

`while (`*`condition`*`)` *`statement`*

Arguments

`condition`

Any Gamma or Lisp expression.

`statement`

Any Gamma or Lisp statement.

Returns

The value of *`condition`* at the final iteration.

Description

This function iterates until its *`condition`* evaluates to `nil`, evaluating the *`statement`* at each iteration. The *`condition`* is evaluated before the *`statement`*, so it is possible for a `while` loop to iterate zero times.

Example

```
Gamma> x = 0;
0
Gamma> while (x < 5) { princ (x, "\n"); x++; }
0
1
2
3
4
4
Gamma> x;
5
Gamma>
```

See Also

`for`, `if`, Statements

with

`with` — traverses an array or list performing a statement.

Syntax

```
with symbol in|on s_exp do statement

with symbol1 in|on s_exp symbol2 = collect|tcollect statement
```

Arguments

`symbol`

Any Gamma or Lisp symbol.

`s_exp`

Any Gamma or Lisp expression.

`statement`

Any statement.

Returns

`nil` when using the `do` option, and the result of the *`statement`* when using the `collect` or `tcollect` option.

Description

`with` *`symbol`* `in|on` *`s_exp`* `do` *`statement`*

- A `with` loop using the iteration style `in` traverses an array or list as defined by an expression (*`s-exp`*), performing the *`statement`* with the iteration *`symbol`* assigned to each element of the array or list in turn.
- A `with` loop using the iteration style `on` traverses a list defined by an expression (*`s-exp`*), performing the *`statement`* with the iteration *`symbol`* assigned to the car and then successive cdrs of the list.
- The iteration *`symbol`* is local in scope to the `with` statement.

`with` $symbol_1$ `in|on` *`s_exp`* $symbol_2$ `= collect|tcollect` *`statement`*

- A `with` loop using the `collect` directive will collect the results of the *`statement`* for each iteration, and produce an array or list (depending on the type of the original array or list), whose elements correspond on a one-to-one basis with the elements of the original array or list.
- A `with` loop using the `tcollect` directive will collect the results of the *`statement`* at each iteration, ignoring `nil` results in the body. The resulting array or list will not have a one-to-one correspondence with the original array or list.
- The result of a `with` loop using `collect` or `tcollect` will be assigned to $symbol_2$, which is not local in scope to the `with` statement. The iteration $symbol_1$ is local in scope to the `with` statement.

Examples

```
Gamma> A = array(1,2,3,4);
[1 2 3 4]
Gamma> with x in A do
{
x = x + 1;
princ(x, "\n");
}
2
3
4
5
nil

Gamma>

Gamma>  L = list (1, 2, 3, 4, 5, 6);
(1 2 3 4 5 6)
Gamma> with x on L do (princ(x,"\n"));
(1 2 3 4 5 6)
(2 3 4 5 6)
(3 4 5 6)
(4 5 6)
(5 6)
(6)
nil
nil
Gamma> with x in L y = collect x + 2;
(3 4 5 6 7 8)
Gamma> y;
(3 4 5 6 7 8)
Gamma> with x in L y = tcollect x < 4 ? x : nil;
(1 2 3)
Gamma> y;
(1 2 3)
Gamma>
```

See Also

for, if, Statements

IV. Core Functions

Table of Contents

call....60
class_add_cvar....61
class_add_ivar....62
class_name....63
class_of....64
defclass....65
defmacro, defmacroe....66
defun, defune,....67
defmethod....68
defvar....69
destroy....70
eq, equal....71
error....73
eval....74
eval_list....75
eval_string....76
force, forceq, forceqq....77
funcall....78
function_args....79
function_body....80
function_name....81
getprop....82
has_cvar....83
has_ivar....84
instance_vars....85
is_class_member....86
ivar_type....87
macro....88
new....90
parent_class....91
print_stack....92
properties....93
quote, backquote....94
require, load....95
set, setq, setqq....97
setprop....98
setprops....99
trap_error....100
unwind_protect....101

whence **102**

call

`call` — calls a class method for a given instance.

Syntax

```
call (instance, method, !argument...)
call (instance, class, method, !argument...)
```

Arguments

`instance`

An instance of a class.

`method`

A method name defined for the class of the instance.

`class`

A class name.

`argument`

The arguments to the method.

Returns

The result of calling the named method on the given instance with the provided arguments.

Description

This function explicitly calls a class method for the provided instance, using the same argument list as would be required for a call using `(instance method ...)` syntax in Lisp, or the `instance.method (...)` syntax in Gamma. The second syntax of this function provides a means for calling an explicit class method even if the class of the instance overloads the method name. Notice that the arguments to call are all evaluated.

Example

This example is based on the class and method developed in `method`.

```
Gamma> call(sqB, Square, #perimeter);
12
Gamma> call(sqB, Square, #area);
9
Gamma>
```

See Also

`method`

class_add_cvar

`class_add_cvar` — adds new class variables.

Syntax

```
class_add_cvar (class variable init_value? type?);
```

Arguments

`class`

The class receiving the new class variable.

`variable`

The new variable to add to the class.

`init_value`

Optional argument for initial value of the variable.

`type`

Optional argument for the type of variable.

Returns

The value of the new argument.

Description

This function adds new class variables to either binary (built-in) or user-defined classes. Class variables are special variables that are available to be set/read by any instance of the class, as well as any derived classes or their instances, whether they were created before or after the class variable was defined.

Example

This example is based on the class and method developed in `method`.

```
Gamma> class_add_cvar(RegPolygon, #linethickness, 2);
2
Gamma> polyD = new(RegPolygon);
{RegPolygon (length) (sides)}
Gamma> polyD.linethickness;
2
Gamma> sqB.linethickness;
2
Gamma>
```

See Also

`class_add_ivar`

class_add_ivar

`class_add_ivar` — adds an instance variable to a class.

Syntax

`class_add_ivar (`*`class`*`, `*`variable`*`, `*`init_value`*`?, `*`type`*`?)`

Arguments

`class`

A class.

`variable`

A symbol to be used as the instance variable name.

`init_value`

Any Gamma or Lisp expression to be used as an initial value.

`type`

An integer number which will be stored with the instance variable. This type is not used by the interpreter, and may be anything.

Returns

The value of the *`init_value`*.

Description

This function dynamically adds an instance variable to a class. All instances of that class which are created after this call will contain this instance variable. All instances created before this call will not contain this instance variable. This function is typically called on a class before any instances are created. It is too late to call this function within an instance constructor. All subclasses of this class will inherit the new instance variable. If the ivar already exists on the class, the only effect of this function is to change the default value.

Example

This example is based on the class and method developed in `method`. The instance sqB does not have "color" as an instance variable because it was created before the instance variable "color" was added.

```
Gamma> class_add_ivar(Square, #color, "red", 12);
"red"
Gamma> sqC = new(Square);
{Square (color . "red") (length) (sides . 4)}
Gamma> sqB;
{Square (length . 3) (sides . 4)}
Gamma>
```

See Also

`instance_vars`

class_name

class_name — gives the name of the class.

Syntax

class_name (*class*|*instance*)

Arguments

class|*instance*

A class or instance of a class.

Returns

The name of the class, as a symbol.

Example

This example is based on the class and method developed in method.

```
Gamma> y = Square;
(defclass Square RegPolygon [(area . (defun Square.area (self) (sqr (@ self length))))][length (sides . 4
Gamma> class_name(y);
Square
Gamma> box = new(Square);
{Square (length) (sides . 4)}
Gamma> class_name(box);
Square
Gamma>
```

See Also

class_of

class_of

class_of — gives the class definition of a given instance.

Syntax

class_of (*instance*)

Arguments

instance

An instance of a class.

Returns

The class of the *instance*.

Description

This function returns the class definition of the *instance*. If the *instance* belongs to a derived class, the most precise class definition is returned (the class which was used to create the instance through a call to new).

Example

This example is based on the class and method developed in method.

```
Gamma> class_of(sqB);
(defclass Square RegPolygon [(area . (defun Square.area (self) (sqr (@ self length))))][length (sides . 4)
Gamma>
```

See Also

class_name

defclass

`defclass` — is the function equivalent of the `class` statement.

See

`class`

defmacro, defmacroe

`defmacro, defmacroe` — are the Lisp equivalents of the `macro` function.

Syntax

```
defmacro (!name, !args, !expression...)
defmacroe (name, args, expression...)
```

This version of the `macro` function is only supported by Lisp. See `macro`.

`defun`, `defune`,

`defun, defune` — are the function equivalents of the `function` statement.

See

`function`

defmethod

`defmethod` — is the function equivalent of the `method` statement.

See

`method`

defvar

`defvar` — defines a global variable with an initial value.

Syntax

```
defvar (!symbol, value, constant_p?)
```

Arguments

`symbol`

A variable name which has not yet been assigned a value.

`value`

Any s_exp.

`constant_p`

If non-`nil`, the symbol will be assigned as a constant.

Returns

The value of the symbol.

Description

This function defines a global variable with an initial value. If the *`constant_p`* argument is present and non-`nil`, then the *`symbol`* becomes a constant, and any attempt to set its value in any scope will fail. If the *`symbol`* already has a value and *`constant_p`* is non-`nil` or absent, then `defvar` will return immediately with the current value of the *`symbol`*. If *`constant_p`* is non-`nil` and the *`symbol`* already has a value, then an error is generated.

The intent of `defvar` is to provide a value for a symbol only if that symbol has not yet been defined. This allows a Gamma or Lisp file to contain default symbol values which may be overridden before the file is loaded.

Example

```
Gamma> defvar(a,7,t);
7
Gamma> a;
7
Gamma> a = 5;
Assignment to constant symbol: a
debug 1>

Gamma> b = 9;
9
Gamma> defvar(b,10);
9
Gamma> b;
9
Gamma>
```

See Also

`set`

destroy

`destroy` — destroys a class instance.

Syntax

```
destroy (instance)
```

Arguments

`instance`

An instance of any class.

Returns

`t` when successful, else error.

Description

This function destroys instances of classes. When a class instance is destroyed, its data type changes to *destroyed instance*. You can test for a destroyed instance by using the predicate `destroyed_p`.

Example

```
Gamma> class RegPolygon{sides; length;}
(defclass RegPolygon nil [][length sides])
Gamma> polyA = new(RegPolygon);
{RegPolygon (length) (sides)}
Gamma> destroy (polyA);
t
Gamma> polyA;
#<Destroyed Instance>
Gamma> destroyed_p(polyA);
t
Gamma>
```

See Also

`class`, `new`

eq, equal

`eq, equal` — compare for identity and equivalence.

Syntax

```
eq (s_exp, s_exp)
equal (s_exp, s_exp)
```

Arguments

s_exp

Any Gamma or Lisp expression.

Returns

`eq` returns `t` if the two arguments are exactly the same Gamma or Lisp element, otherwise `nil`. `equal` returns `t` if the two arguments "look" the same but are not necessarily pointers to the same memory.

Description

The interpreter's storage mechanism allows a particular element to be referenced from more than one location. Functions like `cons` and `list` do not copy their arguments, but simply construct a higher level entity (in these cases a list) which refers to their arguments. The `copy` function will create a new top-level structure but maintain references to the sub-elements of the original list. The `eq` function tests to see whether two elements are in fact references to the same element. The `equal` function determines whether two elements have identical contents, but are not necessarily references to the same element. All things which are eq are also equal. Things which are equal are not necessarily eq.

The `equal` function will travel lists, arrays and instances to compare sub-elements one at a time. The two elements will be equal if all of their sub-elements are equal. Numbers are compared based on actual value, so that `equal(3, 3.0)` is `t`. Strings are compared using `strcmp`.

Symbols are always unique. A symbol is always `eq` to itself.

Example

```
Gamma> a = #acme;
acme
Gamma> b = #acme;
acme
Gamma> equal(a,b);
t
Gamma> eq(a,b);
t

Gamma> a = "acme";
"acme"
Gamma> b = "acme";
"acme"
Gamma> equal(a,b);
t
Gamma> eq(a,b);
nil

Gamma> equal(5,5);
t
Gamma> eq(5,5);
nil
```

```
Gamma> x = list(#acme, list(1,2,3), "hi");
(acme (1 2 3) "hi")
Gamma> y = copy (x);
(acme (1 2 3) "hi")
Gamma> equal(x,y);
t
Gamma> eq(x,y);
nil
Gamma> equal(cadr(x),cadr(y));
t
Gamma> eq(cadr(x),cadr(y));
t
Gamma>
```

See Also

Comparison Operators

error

`error` — redirects the interpreter.

Syntax

```
error (string)
```

Arguments

`string`

A string.

Returns

This function does not return.

Description

The `error` function causes the interpreter to immediately stop what it is doing and to jump to the innermost `trap_error`, `unwind_protect`, interactive session or interprocess communication event handler. The value of `_last_error_` is set to the argument string.

Example

This function will return its argument if the argument is a number, or generate an error and never return if the argument is not a number.

```
function check_number (n)
{
    if (!number_p(n))
 {
          error(string(n," is not a number."));
 }
    n;
}
```

This statement will immediately cause an error if the user presses **`Ctrl-C`** at the keyboard. This is useful for breaking a running program and going to a debugging prompt.

```
signal (SIGINT, #(error ("Keyboard Interrupt")))
```

See Also

`trap_error`, `unwind_protect`

eval

`eval` — evaluates an argument.

Syntax

`eval (`*`s_exp`*`)`

Arguments

s_exp

Any Gamma or Lisp expression.

Returns

The result of evaluating the argument. Note that the argument is also evaluated as part of the function calling mechanism.

Description

The `eval` function forms the basis for running a Gamma program. Every data type has a defined behavior to the `eval` call. These are:

- **symbol** Look up the symbol in the current scope and return the value bound to the symbol. If the symbol is not bound, generate an "Undefined symbol" error.
- **list** Evaluate the first element of the list. If the result is a function, call that function with the rest of the list as arguments. If the first element evaluates to an instance of a class, look up the second element as the method name and resolve that method name in the class or its ancestors. Call the method with the instance bound to self and all other list elements as arguments.
- **all others** All other data types evaluate to themselves.

The `eval` function can be useful when constructing code which must be conditionally executed at a later date, and passed about as data until that time. It may be useful to provide a piece of code as an argument to a generic function so that the function can evaluate it as part of its operation.

Example

Note: The # operator is used to protect an expression from evaluation. See Quote Operators for more information.

```
Gamma> a = 5;
5
Gamma> b = #a;
a
Gamma> b;
a
Gamma> eval(b);
5
Gamma>
```

See Also

`eval_list`

eval_list

`eval_list` — evaluates each element of a list.

Syntax

`eval_list (`*`list`*`)`

Arguments

`list`

A list.

Returns

A new list whose elements are the results of evaluating each of the elements of the argument *`list`* in turn.

Description

Evaluates each element of the *`list`*. Returns the results as a new list whose elements correspond on a one-to-one basis with the elements of the *`list`*.

Example

The # operator is used to protect an expression from evaluation. See Quote Operators for more information.

```
Gamma> a = 5;
5
Gamma> b = 3;
3
Gamma> c = list (#a, #b, "Their sum", #(a+b));
(a b "Their sum" (+ a b))
Gamma> eval_list(c);
(5 3 "Their sum" 8)
Gamma>
```

See Also

`eval`

eval_string

`eval_string` — evaluates a string.

Syntax

`eval_string (`*`string`*`)`

Arguments

`string`

A string.

Returns

The result of evaluating the *`string`* as if it were a Lisp expression.

Description

This function evaluates a string as if it were a Lisp expression, regardless of whether the file syntax is Gamma or Lisp.

Example

```
Gamma> eval_string("(+ 5 6)");
11
Gamma> testvalue = 75;
75
Gamma> eval_string("testvalue");
75
Gamma>
```

force, forceq, forceqq

force, forceq, forceqq — assign a value to a symbol, forcing the evaluation of change functions for the symbol.

Syntax

```
force (symbol, s_exp)
forceq (!symbol, s_exp)
forceqq (!symbol, !s_exp)
```

Arguments

symbol

A symbol.

s_exp

Any Gamma or Lisp expression.

Returns

The *s_exp* argument.

Description

These functions are idential to the set, setq, and setqq, functions, except in addition to assigning a value to a symbol, and being the functional equivalent of the = (assignment) operator, these functions force Gamma to evaluate the change functions for the symbol even if the value has not changed.

This function is particularly useful when working with DataHub points that contain arrays. Gamma handles arrays from the DataHub by mapping them automatically into Gamma arrays, so you can address individual elements. However, in Gamma, if you have a DataHub array point, represented as `$default:myarray`, you can modify an element of the array normally, such as `$default:myarray[0] = 17`; but that does not automatically write back to the DataHub, so nothing gets updated. You have to rewrite the point. Logically you would do this: `$default:myarray = $default:myarray`; to reassign the point. But this is a null operation since you are just assigning the same value again to the point. Using `force`, `forceq`, `forceqq` like this: `force(#$default:myarray, $default:myarray);` forces the point change to be sent back to the DataHub.

The `force` function evaluates both of its arguments, `forceq` evaluates only its second argument, and `forceqq` evaluates neither of its arguments. A symbol's value is the value returned as a result of evaluating that symbol. Symbols constitute the Lisp mechanism for representing variables. These functions can only affect the value of a symbol in the current scope.

See Also

Assignment Operators, set, setq, setqq

funcall

`funcall` — provides compatibility with other Lisp dialects.

Syntax

`funcall (`*`function`*`, `*`args`*`)`

Arguments

`function`

A function definition.

`args`

The arguments to the function.

Returns

The result of the function.

Description

This is provided for compatibility with some other dialects of Lisp. Gamma's version of Lisp, SCADALisp, does not need this function as function definitions can be bound directly to any symbol and called by naming that symbol.

Occasionally this function can use useful in Gamma if a large number of variable arguments are being passed to a function. The called function is named as the first argument and the list of arguments to pass to it are passed as a list in the second arg.

Example

```
Gamma> funcall(atan2, list(5,3));
1.0303768265243125057
Gamma> function plus6 (a,b,c,d,e,f) a+b+c+d+e+f;
(defun plus6 (a b c d e f) (+ (+ (+ (+ (+ a b) c) d) e) f))
Gamma> funcall(plus6, list(1,2,3,4,5,6));
21
Gamma>
```

See Also

`defun`, `function`

function_args

`function_args` — lists the arguments of a function.

Syntax

`function_args (`*`function`*`)`

Arguments

`function`

Any function name.

Returns

A list of the arguments of *`function`*.

Description

This function lists the arguments of any function. Each argument is returned in the form of an association list, whose first element is the function argument, and whose second element represents the argument modifier(s), if any. The hex numbers that correspond to function modifiers are as follows:

- **`0x20000000`** Optional (?).
- **`0x40000000`** Variable length argument (...).
- **`0x80000001`** Not evaluated (!).

Example

```
Gamma> function_args(getprop);
((symbol 0) (property 0))
Gamma> function_args(drain);
((file 0) (t_or_nil 0))
Gamma> function_args(gc);
nil
Gamma> function_args(defvar);
((symbol -2147483648) (value 0) (constant? 536870912))
Gamma> function_args(read);
((file 0))
Gamma> function g (a,b,c) {((a * b)/c);}
(defun g (a b c) (/ (* a b) c))
Gamma> function_args(g);
((a 0) (b 0) (c 0))
Gamma>
```

See Also

`function_body`, `function_name`

function_body

`function_body` — gives the body of a user-defined function.

Syntax

`function_body (`*`function`*`)`

Arguments

`function`

Any function.

Returns

The function definition in Lisp syntax if the function is user-defined, else `nil`.

Description

This function shows the body of a user-defined function in Lisp syntax.

Example

```
Gamma> function g(a,b,c) {(a * b)/c;}
(defun g (a b c) (/ (* a b) c))
Gamma> function_body(g);
#0=((/ (* a b) c))
Gamma> function h(r,s) {sin(r)/cos(s) * tan(s);}
(defun h (r s) (* (/ (sin r) (cos s)) (tan s)))
Gamma> function_body(h);
#0=((* (/ (sin r) (cos s)) (tan s)))
Gamma> function_body(sort);
nil
Gamma>
```

See Also

`function_args`, `function_name`

function_name

`function_name` — gives the name of a function.

Syntax

`function_name (`*`function`*`)`

Arguments

`function`

Any function.

Returns

The name of the function.

Example

```
Gamma> function grand(a,b,c) {(a * b)/c;}
(defun grand (a b c) (/ (* a b) c))
Gamma> function_name(grand);
grand
Gamma> s = grand;
(defun grand (a b c) (/ (* a b) c))
Gamma> function_name(s);
grand

Gamma> function f () { nil; }
(defun f () nil)
Gamma> function g (x) { princ (function_name(x), "\n"); }
(defun g (x) (princ (function_name x) "\n"))
Gamma> g (f);
f
t
Gamma>
```

See Also

`function_args`, `function_body`

getprop

`getprop` — returns a property value for a symbol.

Syntax

`getprop (symbol, property)`

Arguments

`symbol`

A symbol.

`property`

A symbol naming the property to be fetched.

Returns

The value of the property for the given symbol, or `nil` if the property is not defined.

Description

Return the value of the property for the given symbol. Once a property has been set for a symbol, it will remain as long as the Gamma program is running.

Example

```
Gamma> tag001 = 5.5;
5.5
Gamma> setprop(#tag001, #maxlimit,10);
nil
Gamma> getprop(#tag001, #maxlimit);
10
Gamma> getprop(#tag001, #minlimit);
nil
Gamma>
```

See Also

`properties`, `setprop`, `setprops`

has_cvar

has_cvar — queries for the existence of a class variable.

Syntax

has_cvar (*instance|class*, *variable*)

Arguments

instance|class

An instance of a class; or a class.

variable

An class variable name, as a symbol.

Returns

t if any instance, class or any parent (base) class contains the variable, otherwise nil.

Description

This function checks for the existence of class variables for a class or instance of a class. It searches all parent (base) classes.

Example

This example is based on the class and method developed in method and class_add_cvar. The variable name is preceded by # to prevent evaluation. See Quote Operators for more information.

```
Gamma> RegPolygon;
#0=(defclass RegPolygon nil [(linethickness . 2) (perimeter . (defun RegPolygon.perimeter (self) (* (@ se
Gamma> polyD;
{RegPolygon (length) (sides)}
Gamma> Square;
(defclass Square RegPolygon [(area . (defun Square.area (self) (sqr (@ self length))))][length (sides . 4
Gamma> sqB;
{Square (length) (sides . 4)}
Gamma> has_cvar(RegPolygon, #linethickness);
t
Gamma> has_cvar(polyD, #linethickness);
t
Gamma> has_cvar(Square, #linethickness);
t
Gamma> has_cvar(sqB, #linethickness);
t
Gamma>
```

See Also

class_add_cvar

has_ivar

has_ivar — queries for the existence of an instance variable.

Syntax

```
has_ivar (instance|class, variable)
```

Arguments

instance|*class*

An instance of a class; or a class.

variable

An instance variable name, as a symbol.

Returns

t if the instance or class contains the named instance variable, or if any parent (base) of the class contains the instance variable, otherwise nil.

Description

This function queries an instance or class to determine whether a given instance variable exists for that instance or class. When querying classes, if any parent (base) of that class contains the given instance variable, this function returns t. It is possible for a class to contain an instance variable, and an instance of that class not to contain it, but only if class_add_ivar was called after the instance was created. See class_add_ivar for details.

Example

This example is based on the class and method developed in method. The variable name is preceded by # to prevent evaluation. See Quote Operators for more information.

```
Gamma> Square;
(defclass Square RegPolygon [(area . (defun Square.area (self) (sqr (@ self length))))][length (sides . 4
Gamma> sqB;
{Square (length) (sides . 4)}
Gamma> has_ivar(Square, #sides);
t
Gamma> has_ivar(Square, #perimeter);
nil
Gamma> has_ivar(sqB, #area);
nil
Gamma> has_ivar(sqB, #length);
t
Gamma>
```

See Also

class_add_ivar

instance_vars

`instance_vars` — finds all the instance variables of a class or instance.

Syntax

```
instance_vars (instance|class)
```

Arguments

instance|*class*

An instance of a class, or a class.

Returns

An array of all instance variables defined for the given *instance* or *class*. If an instance is queried, then the values of all instance variables for that instance are also reported.

Description

Queries the instance variables of a class or instance.

Example

This example is based on the class and method developed in `method`, `class_add_ivar` and `class_add_cvar`.

```
Gamma> polyD;
{RegPolygon (length) (sides)}
Gamma> sqB;
{Square (length) (sides . 4)}
Gamma> instance_vars(RegPolygon);
[length sides]
Gamma> instance_vars(polyD);
[(length) (sides)]
Gamma> instance_vars(Square);
[length (sides . 4)]
Gamma> instance_vars(sqB);
[(length) (sides . 4)]
Gamma>
```

See Also

`class`, `class_add_cvar`, `class_add_ivar`

is_class_member

`is_class_member` — checks if an instance or class is a member of a class.

Syntax

```
is_class_member (instance|class, class)
```

Arguments

instance|*class*

An instance of a class; or a class.

class

A class.

Returns

`t` if the *instance* or *class* is a member of the *class*, else `nil`.

Description

This function checks if a given instance or class is a member (an instance or derived class) of another class.

Example

This example is based on the classes developed in class.

```
Gamma> sqB = new(Square);
{Square (length) (sides . 4)}
Gamma> is_class_member(sqB, Square);
t
Gamma> is_class_member(Square, RegPolygon);
t
Gamma> is_class_member(sqB, RegPolygon);
t
Gamma> polyF = new(RegPolygon);
{RegPolygon (length) (sides)}
Gamma> is_class_member(polyF, Square);
nil
Gamma>
```

See Also

`new`

ivar_type

`ivar_type` — returns the type of a given instance variable.

Syntax

```
ivar_type (instance, variable)
```

Arguments

`instance`

A class instance.

`variable`

An instance variable name, as a symbol.

Returns

`nil` if the instance does not contain the variable, or the instance variable type, as assigned by `class_add_ivar`.

Description

This function returns the instance variable type for a given instance variable. The instance variable type is not used internally by the Gamma or Lisp engine.

Example

```
Gamma> ivar_type(Osinfo,#cpu);
1
```

See Also

`class_add_ivar`

macro

`macro` — helps generate custom functions.

Syntax

```
macro name (args) statement
```

Arguments

`name`

The name of the macro.

`args`

An argument list.

`statement`

The body of the macro.

Returns

A named macro definition in Lisp syntax.

Description

This function lets Gamma generate custom functions. The most common type of macro is one that will call different functions for different kinds of arguments. Once the macro has been called on a specific kind of argument, successive calls to the macro for that kind of argument will not be processed by the macro at all, but will be handed straight over to its corresponding function.

One advantage is speed, as the macro code is only executed once. Thereafter only the corresponding function code is executed.

Example

1. Define a macro. This macro checks its arguments to see if they are symbols or strings, and performs correspondingly different operations on them.

```
macro symbol_number (!x,!y)
{
   if (symbol_p(x) && symbol_p(y))
      string(string(x),string(y));
   else if (symbol_p(x) && number_p(y))
      ‘setq(@x,@y);
   else if (number_p(x) && symbol_p(y))
      ‘setq(@y,@x);
   else if (number_p(x) && number_p(y))
      y + x;
   else
      error(string("Error: I accept only symbols and numbers."));
}
```

2. Calling the macro gives these results:

```
Gamma> symbol_number(st,art);
"start"
Gamma> symbol_number(myvalue,35);
35
Gamma> myvalue;
```

```
35
Gamma> symbol_number(40,yourvalue);
40
Gamma> yourvalue;
40
Gamma> symbol_number(35,40);
75
Gamma>
```

3. Define a function that includes the macro, and then call that function.

```
Gamma> function f(x,y) {symbol_number(b,3);}
(defun f (x y) (symbol_number b 3))
Gamma> f(#r,7);
3
Gamma>
```

4. Check the function definition. Note that the macro code is now gone. In its place is `setq`, the function it calls for the specified kind of argument.

```
Gamma> f;
(defun f (x y) (setq b 3))
Gamma>
```

See Also

function

new

`new` — creates a new instance of a class.

Syntax

`new (class)`

Arguments

`class`

The name of an existing class.

Returns

A new instance of the *`class`*.

Description

The `new` function creates a new instance of the specified class and initializes any instance variables which have default values associated with them, or assigns them to `nil` if there is no default specified.

An instance is represented by an open brace, followed by the class name, followed by a sequence of dotted pairs (dotted lists of two elements), each containing an instance variable name and a value, followed by a closing brace. Note that `(x . nil)` is the same as `(x)`. For example, an object of the class `PhPoint` would be `{PhPoint (x . 5) (y . 0)}`.

An instance can be destroyed by `destroy`.

Example

```
Gamma> class RegPolygon{sides; length;}
(defclass RegPolygon nil [][length sides])
Gamma> class Square RegPolygon {sides = 4;}
(defclass Square RegPolygon [][length (sides . 4)])
Gamma> polyA = new(RegPolygon);
{RegPolygon (length) (sides)}
Gamma> sqC = new(Square);
{Square (length) (sides . 4)}
Gamma>
```

See Also

`class`, `destroy`

parent_class

`parent_class` — returns the closest parent (base) of a class or instance.

Syntax

`parent_class (`*`instance`*`|`*`class`*`)`

Arguments

`instance``|`*`class`*

An instance of a class; or a class.

Returns

The closest parent (base) class of the *`instance`* or *`class`*.

Description

This function returns the closest (immediate) parent (base) class of the class or instance provided.

Example

```
Gamma> class RegPolygon{sides; length;}
(defclass RegPolygon nil [][length sides])
Gamma> class Square RegPolygon {sides = 4;}
(defclass Square RegPolygon [][length (sides . 4)])
Gamma> class BigSquare Square {length = 30;};
(defclass BigSquare Square [][(length . 30) (sides . 4)])
Gamma> polyA = new(RegPolygon);
{RegPolygon (length) (sides)}
Gamma> sqC = new(Square);
{Square (length) (sides . 4)}
Gamma> bigD = new(BigSquare);
{BigSquare (length . 30) (sides . 4)}
Gamma> parent_class(polyA);
nil
Gamma> parent_class(sqC);
(defclass RegPolygon nil [][length sides])
Gamma> parent_class(bigD);
(defclass Square RegPolygon [][length (sides . 4)])
Gamma> parent_class(Square);
(defclass RegPolygon nil [][length sides])
Gamma> parent_class(BigSquare);
(defclass Square RegPolygon [][length (sides . 4)])
Gamma>
```

See Also

`class`

print_stack

`print_stack` — prints a Gamma stack.

Syntax

```
print_stack (file?, stack)
```

Arguments

`file`

The name of a file.

`stack`

The stack you wish to print.

Returns

`t` if successful, else `nil`.

Description

This function causes Gamma to print a stack, such as `_error_stack_`, `_eval_stack_`, `_jump_stack_`, or `_unwind_stack_`. See Predefined Symbols for more details about these.

Example

```
Gamma> try (2 + nil); catch print_stack(_eval_stack_);
trap_error + print_stack
t
Gamma>
```

See Also

Predefined Symbols

properties

`properties` — should never be used.

Syntax

```
properties (symbol)
```

Arguments

`symbol`

Any symbol.

Returns

The property list for the given symbol.

Description

This function should never be used. Property lists are designed to be handled by the `getprop` and `setprop` functions. Property lists may be represented internally by a number of mechanisms, so the type and structure of the return from this function may change at any time.

See Also

`getprop`, `setprop`, `setprops`

quote, backquote

`quote, backquote` — correspond to Quote Operators.

Syntax

```
quote (s_exp)
backquote (s_exp)
```

Arguments

s_exp

Any Gamma or Lisp expression.

Returns

The *s_exp*, without evaluation.

Description

These are the functional equivalents of the Quote Operators. `quote` is identical to the # quote operator. `backquote` is identical to the ‘ quote operator.

See

Quote Operators

require, load

`require, load` — load files.

Syntax

```
require (filename)
load (filename)
require_lisp (filename)
load_lisp (filename)
required_file (filename)
_require_path_ (filename)
```

Arguments

filename

The name of a file, as a string.

Returns

The name of the file, as a string.

Description

The `require` function loads the named file the first time that it is called. Subsequent calls to `require` with the same filename will simply be ignored. This provides a means for specifying dependencies for applications containing multiple files.

The `load` function loads the named file every time it is called. It attempts to open the named file, read expressions and evaluate them one at a time until the end of file is reached. `load` attempts to find the file by prepending each of the entries in `_require_path_` to the filename. If the file is not found, then `load` appends each of the entries in `_load_extensions_` to the path resulting from concatenating `_require_path_` and filename. If the file is still not found, `nil` is returned. If a different grammar has been defined for the loader, then that grammar will be used to read the file.

`require_lisp` and `load_lisp` operate similarly to `require` and `load`, except they treat any file as a Lisp file. This is helpful when using Lisp libraries with alternate grammars such as Gamma or user-defined grammars.

`required_file` determines which file would be loaded as the result of a call to `require` or `require_lisp`, but does not actually load it. This can be useful in debugging to determine where a particular function or file is coming from.

The pre-defined global variable `_require_path_` contains a list of the paths to be searched to find the specified filename. This variable is initialized to ("" "/usr/cogent/lib"), which references the current directory and the standard location for cogent libraries. The list of paths can be augmented with:

```
_require_path_ = cons ("my_directory_name", _require_path_);
```

Example

```
Gamma> require("x/myfile.dat");
"x/myfile.dat"
Gamma> require("x/myfile.dat");
t
Gamma> load("x/myfile.dat");
"x/myfile.dat"
```

```
Gamma> required_file("x/myfile.dat");
t
Gamma> require_lisp("myfileli.dat");
nil
Gamma>
```

See Also

Loading Files

set, setq, setqq

set, setq, setqq — assign a value to a symbol.

Syntax

```
set (symbol, s_exp)
setq (!symbol, s_exp)
setqq (!symbol, !s_exp)
```

Arguments

symbol

A symbol.

s_exp

Any Gamma or Lisp expression.

Returns

The *s_exp* argument.

Description

These functions assign a value to a symbol, and are the functional equivalent of the = (assignment) operator. Normally in Gamma the = operator is used for assignment, but these functions give more control over evaluation of symbols and expressions at the point of assignment.

The set function evaluates both of its arguments. setq evaluates only its second argument and setqq evaluates neither of its arguments. The most commonly used of these functions is setq. A symbol's value is the value returned as a result of evaluating that symbol. Symbols constitute the Lisp mechanism for representing variables. These functions can only affect the value of a symbol in the current scope.

Example

```
Gamma> setq(y, 6);
6
Gamma> setq(x, #y);
y
Gamma> set(x, 5);
5
Gamma> x;
y
Gamma> y;
5
Gamma>
```

See Also

Assignment Operators, force

setprop

`setprop` — sets a property value for a symbol.

Syntax

```
setprop (symbol, property, value)
```

Arguments

`symbol`

The symbol whose property will be set.

`property`

A symbol which identifies the property to be set.

`value`

The new value of the property.

Returns

The previous value for that property, or `nil` if there was no previous value.

Description

All symbols in Gamma may have properties assigned to them. These properties are not limited by the scope of the symbol, so that a symbol's property list is always global. A property consists of a (name . value) pair. Property lists are automatically maintained by `setprop` to ensure that each property name is unique for a symbol. A symbol may have any number of properties. A property for a symbol is queried using `getprop`.

The *`symbol`* and *`property`* are normally protected from evaluation when setting properties, using the # operator.

Example

```
Gamma> setprop(#weight,#hilimit,1000);
nil
Gamma> setprop(#weight,#hiwarning,950);
nil
Gamma> setprop(#weight,#lowlimit,500);
nil
Gamma> setprop(#weight,#lowwarning,550);
nil
Gamma> getprop(#weight,#hilimit);
1000
Gamma> getprop(#weight,#lowwarning);
550
Gamma>
```

See Also

`getprop`, `properties`, `setprops`

setprops

`setprops` — lists the most recent property value settings.

Syntax

`setprops (symbol, properties)`

Arguments

`symbol`

The symbol whose properties will be listed.

`properties`

Any property.

Returns

A list of properties with their most recent values, as associated pairs.

Description

This function is used to get a list of all the properties and their associated values as (name . value) pairs. It is called using any of the symbol's properties. The list contains current values in order from the most to least recently entered.

The *`symbol`* and *`property`* are normally protected from evaluation when setting properties, using the # operator.

Example

```
Gamma> setprop(#weight,#hilimit,1000);
nil
Gamma> setprop(#weight,#lowlimit,500);
nil
Gamma> setprop(#weight,#warning,950);
nil
Gamma> setprop(#weight,#warning,975);
950
Gamma> setprops(#weight,#hilimit);
((warning . 975) (lowlimit . 500) (hilimit . 1000))
Gamma>
```

See Also

`setprop`

trap_error

`trap_error` — traps errors in the body code.

Syntax

`trap_error (!body, !error_body)`

Arguments

body

Any Gamma or Lisp expression.

error_body

Any Gamma or Lisp expression.

Returns

The result of the *body*, unless an error occurs during its evaluation, in which case the result of evaluating the *error_body*.

Description

This function traps any errors which occur while evaluating the *body* code. If no error occurs, then `trap_error` will finish without ever evaluating the *error_body*. If an error does occur, `trap_error` will evaluate the *error_body* code immediately and the error condition will be cleared. This is usually used to protect a running program from a piece of unpredictable code, such as an event handler. If the error is not trapped it will be propagated to the top-level error handler where it will cause the interpreter to go into an interactive debugging session.

Example

The following piece of code will run an event loop and protect against an unpredictable event.

```
while(t)
{
 trap_error(next_event(),print_trapped_error());
}

function print_trapped_error ()
{
 princ("Error\n", _error_stack_, "\n occurred...\n");
 princ("Clearing error condition and continuing.\n");
}
```

See Also

`error`, `unwind_protect`, `try catch`

unwind_protect

`unwind_protect` — ensures code will be evaluated, despite errors in the body code.

Syntax

`unwind_protect (!`*`body`*`, !`*`protected_body`*`)`

Arguments

`body`

Any Gamma or Lisp expression.

`protected_body`

Any Gamma or Lisp expression.

Returns

The result of evaluating the *`protected_body`* code. If an error occurs then this function does not return.

Description

This function ensures that a piece of code will be evaluated, even if an error occurs within the *`body`* code. This is typically used when an error might occur but cleanup code has to be evaluated even in the event of an error. The error condition will not be cleared by this function. If an error occurs then control will be passed to the innermost `trap_error` function or to the outer level error handler immediately after the *`protected_body`* is evaluated.

Example

The following code will close its file and run a `write_all_output` function even if an error occurs.

```
if (fp=open("filename","w"))
{
 unwind_protect(write_all_output(),close(fp));
}
```

See Also

`error`, `trap_error`, `protect unwind`

whence

`whence` — gives input information.

Syntax

`whence (`*`s_exp`*`)`

Arguments

`s_exp`

Any Gamma or Lisp expression.

Returns

A list whose car is the input source, and whose cdr is an integer showing the sequential input order of the expression.

Description

This function checks the input source and sequence of any given Gamma expression. It returns this information in the form of a list. The sequence number is assigned the first time the expression is used, and does not change. The `whence` call itself is a Gamma expression, and thus generates a sequence number each time it is called.

This function requires Gamma to be running in debugging mode. To start debugging mode, you must include the **`-d`** option when starting Gamma.

Example

```
[~/w/devel/lisp]$ gamma -d
Gamma(TM) Advanced Programming Language
Copyright (C) Cogent Real-Time Systems Inc., 1996.  All rights reserved.
Version 2.4 Build 147 at Sep 13 1999 17:15:51
Gamma> c = 12;
12
Gamma> d = 14;
14
Gamma> whence(c);
("stdin" 1)
Gamma> whence(d);
("stdin" 2)
Gamma> f = 16;
16
Gamma> whence(f);
("stdin" 6)
Gamma>
```

V. Lists and Arrays

Table of Contents

append 104
aref 105
array 106
array_to_list 107
aset 108
assoc, assoc_equal 109
bsearch 110
car, cdr, and others 111
cons 112
copy 113
copy_tree 114
delete 115
difference 116
find, find_equal 117
insert 118
intersection 119
length 120
list, listq 121
list_to_array 122
make_array 123
nappend 124
nremove 125
nreplace, nreplace_equal 126
nth_car, nth_cdr 127
remove 128
reverse 129
rplaca, rplacd 130
shorten_array 131
sort 132
union 133

append

`append` — concatenates several lists into a single new list.

Syntax

`append (`*`list`*`...)`

Arguments

`list`

One or more lists.

Returns

A new list whose elements are the all of the elements in the given lists, in the order that they appear in the argument lists.

Description

This function concatenates all of the argument lists into a single new list, in the order the arguments are given. Each list is appended to the preceding list by assigning it to the cdr of the last element of that list. The appending is non-destructive; for a destructive version of append use `nappend`.

Example

```
Gamma> append (list(1,2,3), list(4,5));
(1 2 3 4 5)
Gamma> append (list(#a,#c,#g), list(#b,#d,#z));
(a c g b d z)
Gamma>
```

See Also

`nappend`

aref

`aref` — returns an array expression at a given index.

Syntax

`aref (`*`array`*`, `*`index`*`)`

Arguments

`array`

An array.

`index`

A number giving the index into the array, starting at zero.

Returns

The array element at the given index in the array.

Description

The index starts at zero, and extends to the length of the array minus one. If the index is not valid in the array, `nil` is returned, but no error is generated.

Note: This function is identical to the square bracket syntax for referencing array elements, with the syntax:

```
array[index]
```

Example

```
Gamma> x = array (1, 5, #d, "farm");
[1 5 d "farm"]
Gamma> aref (x, 0);
1
Gamma> aref (x, 8);
nil
Gamma> x[3];
"farm"
Gamma>
```

See Also

`array`, `aset`, `delete`, `insert`, `sort`

array

`array` — constructs an array.

Syntax

`array (`*`s_exp`*`?...)`

Arguments

`s_exp`

Any Gamma or Lisp expression.

Returns

An array containing all of the arguments.

Description

An array is represented as a sequence of objects surrounded by square brackets, as in [1 2 a (4 5)]. The objects within the brackets are not evaluated. To refer to or access an array, it must be assigned to a symbol.

This function constructs an array of all of the arguments, in the order given. The arguments are evaluated when the `array` function is called, but once the array has been constructed the array objects are not evaluated.

It is possible to create an empty array, and fill it later. It will expand as necessary when array objects are added.

Example

```
Gamma> array(#a, 5, nil, 4 + 3, "goodbye");
[a 5 nil 7 "goodbye"]
Gamma> y = array(5 * 2, #symbol, 432, "string", nil);
[10 symbol 432 "string" nil]
Gamma> z = array(#c, y);
[c [10 symbol 432 "string" nil]]
Gamma> x = array();
[]
Gamma> x[5] = 19;
19
Gamma> x;
[nil nil nil nil nil 19]
Gamma>
```

See Also

`aset`, `aref`, `insert`, `delete`, `sort`

array_to_list

`array_to_list` — converts an array to a list.

Syntax

```
array_to_list (array);
```

Arguments

`array`

Any array.

Returns

The array converted to a list.

Description

This convenience function converts the top level of an array to a list. Lower level arrays in the resulting list will remain unchanged unless converted separately.

Example

```
Gamma> a = array(1,2,3);
[1 2 3]
Gamma> b = array(4,5,6);
[4 5 6]
Gamma> c = array(7,8,9);
[7 8 9]
Gamma> d = array(a,b,c);
[[1 2 3] [4 5 6] [7 8 9]]
Gamma> e = array_to_list(d);
([1 2 3] [4 5 6] [7 8 9])
Gamma> list_p(e);
t
Gamma> list_p(a);
nil
Gamma>
```

See Also

`list_to_array`

aset

`aset` — sets an array element to a value at a given index.

Syntax

```
aset (array, index, value)
```

Arguments

array

An array.

index

A numeric index into the array.

value

The new value to be placed in the array.

Returns

The *value* argument.

Description

Sets an *array* element to the *value* at the *index*. If the index is past the end of the array, then the array will be extended with `nil`s to the index and the *value* inserted.

This function can also be called using square bracket syntax for referencing array elements, with the syntax:

```
array[index]
```

Example

```
Gamma> x = array (3, 5, #b, nil);
[3 5 b nil]
Gamma> aset(x, 3, 7);
7
Gamma> x;
[3 5 b 7]
Gamma> x[0] = 9;
9
Gamma> x;
[9 5 b 7]
Gamma>
```

See Also

`array`, `aref`, `insert`, `delete`, `sort`

assoc, assoc_equal

assoc, assoc_equal — search an association list for a sublist.

Syntax

```
assoc (s_exp, list)
assoc_equal (s_exp, list)
```

Arguments

s_exp

Any Gamma or Lisp expression.

list

An association list.

Returns

A list whose members are the remainder of the association list starting at the element whose car is eq or equal to the *s_exp*.

Description

An association list is a list whose elements are also lists, each of which typically contains exactly two elements. assoc searches an association list for a sublist whose car is eq to the given *s_exp*. assoc_equal uses equal instead of eq for the comparison. If no matching sublist is found, returns nil.

A symbol-indexed association list (or sym-alist) is an association list where the car of each element is a symbol. This is a common construct for implementing property lists and lookup tables. Since symbols are always unique, sym-alists can be searched with assoc instead of assoc_equal.

Example

```
Gamma> a = 10;
10
Gamma> b = 20;
20
Gamma> c = 30;
30
Gamma> x = list (list(a,15), list(b,25), list(c, 35));
((10 15) (20 25) (30 35))
Gamma> assoc (b,x);
((20 25) (30 35))
Gamma> assoc (20,x);
nil
Gamma> assoc_equal(20,x);
((20 25) (30 35))
Gamma>
```

See Also

car, cdr, eq, equal, Data Types and Predicates

bsearch

`bsearch` — searches an array or list for a element.

Syntax

`bsearch (list_or_array, key, compare_function)`

Arguments

`list_or_array`

A list or array whose elements are sorted.

`key`

The list or array element to search for.

`compare_function`

A function used to compare the *`key`* with the array elements.

Returns

An association list composed of the *`key`* and it's position in the array.

Description

This function performs a binary search on an array based on a comparison function you provide. The *`compare_function`* must return a negative number if the value is ordinally less than the list or array element, 0 if the two are equal and a positive number if the value is ordinally greater than the list or array element. The *`array`* or *`list`* must be sorted in an order recognizable by the *`compare_function`* for this function to work.

Example

```
Gamma> function comp (x,y) {x - y;}
(defun comp (x y) (- x y))
Gamma> Ax = array(9,2,11,31,13,8,15,95,17,5,19,6,21);
[9 2 11 31 13 8 15 95 17 5 19 6 21]
Gamma> Sx = sort(Ax,comp);
[2 5 6 8 9 11 13 15 17 19 21 31 95]
Gamma> bsearch(Sx,19,comp);
(19 . 9)
Gamma> bsearch(Sx,5,comp);
(5 . 1)
Gamma>
```

See Also

sort

car, cdr, and others

car, cdr, and others — return specific elements of a list.

Syntax

```
car (list)
cdr (list)
caar (list)
cadr (list)
cdar (list)
cddr (list)
caaar (list)
caadr (list)
cadar (list)
caddr (list)
cdaar (list)
cdadr (list)
cddar (list)
cdddr (list)
```

Arguments

list

Any list.

Returns

An element of the *list* or nil.

Description

The car function returns the first element of a list. The cdr function returns all of the list except for the first element. The remaining functions in this group are simply shortcuts for the common combinations of car and cdr. The shortcut functions are read from left to right as nested car and cdr calls. Thus, a call to caddr (mylist) would be equivalent to car (cdr (cdr (mylist))). If the argument is not a list, the result is nil. The cdr function will only return a non-list result in the case of a dotted pair.

Example

```
Gamma> car(list(1,2));
1
Gamma> cdr(list(1,2));
(2)
Gamma> cdr(cons(1,2));
2
Gamma> caadr (list (1, list(2, 3, list(4, 5), 6)));
2
Gamma> cdadr (list (1, list(2, 3, list(4, 5), 6)));
(3 (4 5) 6)
Gamma>
```

See Also

cons, list, nth_car, nth_cdr

cons

`cons` — constructs a cons cell.

Syntax

`cons (car_exp, cdr_exp)`

Arguments

`car_exp`

Any Gamma or Lisp expression.

`cdr_exp`

Any Gamma or Lisp expression.

Returns

A list whose car is *`car_exp`* and whose cdr is *`cdr_exp`*.

Description

This function constructs a list whose car and cdr are *`car_exp`* and *`cdr_exp`* respectively. This construction is also known as a *cons cell*. If the *`cdr_exp`* is a list, this has the effect of increasing the list by one member, that is, adding the *`car_exp`* to the beginning of the *`cdr_exp`*.

Example

```
Gamma> a = list(2,3,4);
(2 3 4)
Gamma> b = 5;
5
Gamma> cons(b,a);
(5 2 3 4)
Gamma> cons(a,b);
((2 3 4) . 5)
Gamma> cons(5,nil);
(5)
Gamma>
```

See Also

Data Types and Predicates

copy

`copy` — makes a copy of the top list level of a list.

Syntax

`copy (`*`s_exp`*`)`

Arguments

`s_exp`

Any Gamma or Lisp expression.

Returns

A copy of the top list level of the argument.

Description

This function makes a copy of the top list level of the argument if the argument is a list, otherwise it simply returns the argument. This produces a new list which is equal to the previous list, and whose elements are eq. That is, the elements are not copied but simply reside in both the original and the copy.

Example

```
Gamma> a = list(1, list(2,3,list(4),5));
(1 (2 3 (4) 5))
Gamma> b = copy(a);
(1 (2 3 (4) 5))
Gamma> cadr(a);
(2 3 (4) 5)
Gamma> equal(cadr(a),cadr(b));
t
Gamma> eq(cadr(a),cadr(b));
t
Gamma>
```

See Also

`copy_tree`, `eq, equal`

copy_tree

`copy_tree` — copies the entire tree structure and elements of a list.

Syntax

`copy_tree (`*`s_exp`*`)`

Arguments

`s_exp`

Any Gamma or Lisp expression.

Returns

A copy of the entire tree structure and elements of the argument, if it is a list. Otherwise, the argument.

Description

This function makes a recursive copy of the entire tree structure of the argument if the argument is a list, otherwise it simply returns the argument. This produces a new list which is equal to the previous list, and whose elements are equal, but not eq. That is, the elements are all copied down to the level of the non-list leaves. They are equal to the original elements, but they are different elements. Thus they are equal but not eq.

Example

```
Gamma> a = list(1,list(2,3,list(4),5));
(1 (2 3 (4) 5))
Gamma> b = copy_tree(a);
(1 (2 3 (4) 5))
Gamma> cadr(a);
(2 3 (4) 5)
Gamma> equal(cadr(a),cadr(b));
t
Gamma> eq(cadr(a),cadr(b));
nil
Gamma>
```

See Also

`copy`, `eq, equal`

delete

`delete` — removes an element from an array.

Syntax

```
delete (array, position)
```

Arguments

`array`

An array.

`position`

An integer giving the zero-indexed position of the array element to delete.

Returns

The deleted array element, or `nil` if none was deleted. The return value will also be `nil` if the deleted element was `nil` itself.

Description

This function removes an element from an *`array`* and compresses the rest of the array to reduce its overall length by one. If the position is beyond the bounds of the array, nothing happens. This function is destructive.

Example

```
Gamma> a = [1,2,3,4,5];
[1 2 3 4 5]
Gamma> delete(a,3);
4
Gamma> a;
[1 2 3 5]
Gamma>
```

See Also

`insert`

difference

`difference` — constructs a list of the differences between two lists.

Syntax

`difference (listA, listB)`

Arguments

`listA`

A list.

`listB`

A list.

Returns

All elements in *`listA`* that are not in *`listB`*.

Description

Constructs a new list that contains all of the elements in *`listA`* not contained in *`listB`* as compared by the function `eq`.

Example

```
Gamma> a = 1;
1
Gamma> b = 2;
2
Gamma> c = 3;
3
Gamma> d = 4;
4
Gamma> e = 5;
5
Gamma> A = list (a, b, c);
(1 2 3)
Gamma> B = list (b, d, e, c);
(2 4 5 3)
Gamma> difference (A, B);
(1)
Gamma> difference (B, A);
(4 5)
Gamma>
```

See Also

`eq, equal`, `intersection`, `union`

find, find_equal

find, find_equal — search a list using the eq and equal functions.

Syntax

```
find (s_exp, list)
find_equal (s_exp, list)
```

Arguments

s_exp

Any Gamma or Lisp expression.

list

A list to be searched.

Returns

The tail of the *list* starting at the matching element. If no match is found, nil.

Description

The find function searches the *list* comparing each element to the *s_exp* with the function eq. The find_equal function uses equal instead of eq for the comparison.

Example

```
Gamma> find(#a,#list(d,x,c,a,f,t,l,j));(a f t l j)
Gamma> find("hi", #list("Bob","says", "hi"));
nil
Gamma> find_equal("hi",#list("Bob","says","hi"));
("hi")
Gamma>
```

See Also

eq, equal

insert

`insert` — inserts an array value at a given position.

Syntax

`insert (`*`array`*`, `*`position`*`|`*`compare_function`*`, `*`value`*`)`

Arguments

`array`

An array.

`position`

A number giving the zero-based position of the new element within the array, or a function.

`compare_function`

A function on two arguments used to compare elements in the list or array.

`value`

Any Gamma or Lisp expression.

Returns

The *`value`* inserted.

Description

The `insert` function widens an array at the given *`position`* and inserts the *`value`*. If a *`compare_function`* is used, it must return a negative number if the value is ordinally less than the array element, 0 if the two are equal and a positive number if the value is ordinally greater than the array element. The *`value`* will be inserted using a binary insertion sort, based on the return value of the function.

Example

```
Gamma> x = array("a", "b", "c");
["a" "b" "c"]
Gamma> insert(x,3,"d");
"d"
Gamma> x;
["a" "b" "c" "d"]
Gamma> insert(x,strcmp,"acme");
"acme"
Gamma> x;
["a" "acme" "b" "c" "d"]
Gamma>
```

See Also

aref, array, aset

intersection

`intersection` — constructs a list of all the elements found in both of two lists.

Syntax

`intersection (listA, listB)`

Arguments

`listA`

A list.

`listB`

A list.

Returns

All elements which appear in both *`listA`* and *`listB`*.

Description

This function generates a new list which contains all of the elements that appear in both *`listA`* and *`listB`*. The elements are compared using eq. The order of the elements in the resulting list is not defined.

Example

```
Gamma> A = list(#a,#b,#c);
(a b c)
Gamma> B = list(#b,#c,#d);
(b c d)
Gamma> intersection(A,B);
(b c)
Gamma>
```

See Also

`eq, equal` `difference` `union`

length

`length` — counts the number of elements in a list or array.

Syntax

`length (`*`list`*`)`

Arguments

`list`

A list or array.

Returns

The number of elements in the *`list`* or *`array`*. If the argument is not a list or array, returns 0.

Example

```
Gamma> length(list(#a,#b,#c,#d));
4
Gamma> length([11,13,15,17,19,21]);
6
Gamma> length(sqr(2 + 3));
0
Gamma>
```

list, listq

`list, listq` — create lists.

Syntax

```
list (s_exp?...)
listq (!s_exp?...)
```

Arguments

s_exp

Any Gamma or Lisp expression.

Returns

A list containing all of the arguments.

Description

A list is represented as a sequence of objects surrounded by parentheses, as in (1 2 a [4 5]), possibly with a dot between the second-to-last and last elements in the list. A literal list can be read from a file or from the command line, but must be quoted (using a quote operator) within code to make it literal.

The `list` function creates a list from its arguments. `listq` creates a list from its arguments without evaluation.

Example

```
Gamma> list(4+5,6,"hi",#xref);
(9 6 "hi" xref)
Gamma> listq(4+5,6,"hi",#xref);
((+ 4 5) 6 "hi" 'xref)
Gamma>
```

See Also

Data Types and Predicates Lists and Arrays

list_to_array

`list_to_array` — converts a list to an array.

Syntax

`list_to_array (`*`list`*`)`

Arguments

`list`

A list to convert to an array.

Returns

The *`list`* converted to an array.

Description

This convenience function converts the top level of a list to an array. Sub-lists will remain unchanged unless converted separately.

Example

```
Gamma> a = list(1,2,3);
(1 2 3)
Gamma> b = list(4,5,6);
(4 5 6)
Gamma> c = list(7,8,9);
(7 8 9)
Gamma> d = list(a,b,c);
((1 2 3) (4 5 6) (7 8 9))
Gamma> e = list_to_array(d);
[(1 2 3) (4 5 6) (7 8 9)]
Gamma> array_p(e);
t
Gamma> array_p(a);
nil
Gamma>
```

See Also

`array_to_list`

make_array

`make_array` — creates an empty array.

Syntax

```
make_array (n_elements)
```

Arguments

`n_elements`

A number of elements.

Returns

An array with the given number of elements, all `nil`.

Description

Creates an empty array for later use. This function has become obsolete as the `array` function can now create empty arrays. See `array`.

Example

```
Gamma> make_array(4);
[nil nil nil nil]
Gamma> make_array(7);
[nil nil nil nil nil nil nil]
Gamma>
```

See Also

`array`

nappend

`nappend` — appends one or more lists, destructively modifying them.

Syntax

```
nappend (list...)
```

Arguments

`list`

One or more lists which will be appended in order.

Returns

The first *`list`*, modified in place with the remaining lists appended onto it.

Description

This function appends one or more lists, destructively modifying all but the last argument. It is otherwise identical to `append`.

Example

```
Gamma> a = list (1, 2, 3);
(1 2 3)
Gamma> b = list (4, 5, 6);
(4 5 6)
Gamma> c = list (7, 8, 9);
(7 8 9)
Gamma> nappend (a, b, c);
(1 2 3 4 5 6 7 8 9)
Gamma> a;
(1 2 3 4 5 6 7 8 9)
Gamma> b;
(4 5 6 7 8 9)
Gamma> c;
(7 8 9)
Gamma>
```

See Also

`append`

nremove

nremove — removes list items, destructively altering the list.

Syntax

nremove (*s_exp*, *list*, *use_equal*?)

Arguments

s_exp

Any Gamma or Lisp expression.

list

A list.

use_equal

If non-nil, use equal instead of eq for comparison.

Returns

The *list* with any elements which are eq (or equal if specified) to *s_exp* destructively removed.

Description

This function removes all occurrences of the *s_exp* within the given list and destructively alters the list to reduce its size by one for each occurrence. The default comparison used is eq. If the first argument is removed, then the return value will be (cdr list) with all other occurrences of *s_exp* destructively removed.

Example

```
Gamma> y = list (#a, #b, #c);
(a b c)
Gamma> nremove (#b, y);
(a c)

Gamma> x = list(1,2,3,4,5,6);
(1 2 3 4 5 6)
Gamma> nremove(3, x);
(1 2 3 4 5 6)
Gamma> nremove(3, x, t);
(1 2 4 5 6)

Gamma> y = list(1,2,3,4,1,2,3,4,1,2);
(1 2 3 4 1 2 3 4 1 2)
Gamma> nremove (1,y,t);
(2 3 4 2 3 4 2)
Gamma>
```

See Also

nreplace, remove

nreplace, nreplace_equal

nreplace, nreplace_equal — replace elements in a list.

Syntax

```
nreplace (new_s_exp, old_s_exp, list)
nreplace_equal (new_s_exp, old_s_exp, list)
```

Arguments

new_s_exp

The new expression to be inserted into the list.

old_s_exp

The expression in the list to be replaced.

list

A list.

Returns

The *list*, with all occurrences of *old_s_exp* destructively replaced by *new_s_exp*.

Description

nreplace traverses the *list*, replacing any element which is eq to *old_s_exp* with *new_s_exp*. nreplace_equal uses equal as its comparison function.

Example

```
Gamma> R = list (#f, nil, 5, #ftg);
(f nil 5 ftg)
Gamma> nreplace(#h, #ftg, R);
(f nil 5 h)

Gamma> x = list(1,2,3,1,6,7);
(1 2 3 1 6 7)
Gamma> nreplace(4,1,x);
(1 2 3 1 6 7)
Gamma> nreplace_equal(4,1,x);
(4 2 3 4 6 7)
Gamma> x;
(4 2 3 4 6 7)
Gamma>
```

See Also

remove, nremove

nth_car, nth_cdr

nth_car, nth_cdr — iteratively apply the car and cdr functions to a list.

Syntax

```
nth_car (list, number)
nth_cdr (list, number)
```

Arguments

list

Any list.

number

The number of cars (or cdrs) to apply to the list argument. Non-integers are rounded down. Non-numbers are treated as zero.

Returns

An element of the *list*, or nil.

Description

The nth_car and nth_cdr functions iteratively apply the car and cdr functions to a list. If the list argument is not a list, or if the result of any subsequent application of car or cdr is not a list, the result is nil. If the number of applications is less than or equal to 0, the result is the original list.

Example

```
Gamma> c = list (list(list(list(4,5))));
((((4 5))))
Gamma> nth_car(c,2);
((4 5))
Gamma> nth_car(c,4);
4

Gamma> b = list (6,7,8,9,10);
(6 7 8 9 10)
Gamma> nth_cdr (b,2);
(8 9 10)
Gamma> nth_cdr(b,5);
nil
Gamma> nth_cdr(b,4);
(10)
Gamma>
```

See Also

cons, list, car, cdr

remove

`remove` — removes list items without altering the list.

Syntax

```
remove (s_exp, list, use_equal?)
```

Arguments

s_exp

An expression to remove from the list.

list

The list from which to remove the s_exp.

use_equal

An optional argument. If `t`, `remove` uses the `equal` function for equality, otherwise it uses the more stringent `eq`.

Returns

The *list*, with any matching *s_exp* removed.

Description

This function non-destructively walks a list and removes elements matching the passed *s_exp* using either eq or equal.

Example

```
Gamma> A = list (#a, #b, #c, #b, #a);
(a b c b a)
Gamma> remove (#a, A);
(b c b)
Gamma> A;
(a b c b a)
Gamma> B = list(1,2,3,2,1);
(1 2 3 2 1)
Gamma> remove(2,B);
(1 2 3 2 1)
Gamma> remove(2,B,t);
(1 3 1)
Gamma>
```

See Also

`nremove`

reverse

`reverse` — reverses the order of list elements.

Syntax

`reverse (`*`list`*`)`

Arguments

`list`

A list.

Returns

A new list which whose top-level structure is the reverse of the input *`list`*. If the argument is not a list, returns the argument.

Description

None of the elements of the original *`list`* is copied. The resulting list contains elements which are eq to the corresponding elements in the original. The original *`list`* is not changed.

Example

```
Gamma> S = list(1,2,3);
(1 2 3)
Gamma> R = reverse (S);
(3 2 1)
Gamma> S;
(1 2 3)
Gamma> car (S);
1
Gamma> caddr(R);
1
Gamma> eq (car (S),caddr(R));
t
Gamma>
```

rplaca, rplacd

`rplaca, rplacd` — replace the car and cdr of a list.

Syntax

```
rplaca (cons, s_exp)
rplacd (cons, s_exp)
```

Arguments

list

A list element.

s_exp

Any Gamma or Lisp expression.

Returns

The *s_exp*, or `nil` on failure.

Description

These functions destructively alter the form of a list. `rplaca` modifies the car of a list, effectively replacing the first element. `rplacd` modifies the cdr of a list, replacing the entire tail of the list. These functions have no meaning for non-lists. To entirely remove the tail of a list, replace the cdr of the list with `nil`.

Example

```
Gamma> x = list(1,2,3,4);
(1 2 3 4)
Gamma> rplaca(x,0);
0
Gamma> x;
(0 2 3 4)
Gamma> rplacd(x,list(7,8,9,10,11,12));
(7 8 9 10 11 12)
Gamma> x;
(0 7 8 9 10 11 12)
Gamma>
```

See Also

`car, cdr`

shorten_array

shorten_array — reduces or expands the size of an array.

Syntax

shorten_array (*array*, *size*)

Arguments

array

The array to shorten or expand.

size

The new length for the array.

Returns

The resized array.

Description

This function reduces or expands the size of an array by cutting off any elements which extend beyond the given size, or by adding nils to the end of the array until the new size is reached. This function is analogous to the C function, realloc.

Example

```
Gamma> a = array (1,2,3,4,5);
[1 2 3 4 5]
Gamma> shorten_array(a,3);
[1 2 3]
Gamma> shorten_array(a,9);
[1 2 3 nil nil nil nil nil nil]
Gamma>
```

See Also

array, make_array

sort

`sort` — sorts a list or array, destructively modifying the order.

Syntax

`sort (`*`list_or_array`*`, `*`compare_function`*`)`

Arguments

`list_or_array`

A list or an array.

`compare_function`

A function on two arguments used to compare elements in the list or array.

Returns

The input *`list`* or *`array`*, sorted.

Description

This function sorts the *`list`* or *`array`* in place, destructively modifying the order of the elements. The *`compare_function`* must be a function on two arguments which returns: an integer value less than zero if the first argument is ordinally less than the second, zero if the two arguments are ordinally equal, and greater than zero if the first argument is ordinally greater than the second. This function uses the quicksort algorithm.

Example

```
Gamma> x = list("one","two","three","four","five");
("one" "two" "three" "four" "five")
Gamma> sort(x,strcmp);
("five" "four" "one" "three" "two")
Gamma> x;
("five" "four" "one" "three" "two")
Gamma>
```

union

`union` — constructs a list containing all the elements of two lists.

Syntax

```
union (listA, listB)
```

Arguments

`listA`

A list.

`listB`

A list.

Returns

A new list containing all elements in *`listA`* plus all elements in *`listB`* which do not appear in *`listA`*.

Description

The resulting list will not contain duplicate elements from either list. This function uses eq for comparisons.

Example

```
Gamma> union (list (#j,#j,#j,#k,#l,#j),list(#k,#k,#l,#m,#n));
(j k l m n)
Gamma> union(list(1,2),list(5,1,2,7));
(1 2 5 1 2 7)
Gamma>
```

See Also

`difference`, `intersection`

VI. Strings and Buffers

Table of Contents

bdelete 135
binsert 136
buffer 137
buffer_to_string 138
format 139
make_buffer 141
open_string 142
parse_string 143
raw_memory 145
shell_match 146
shorten_buffer 147
strchr, strrchr 148
strcmp, stricmp 149
string 150
stringc 151
string_file_buffer 152
string_split 153
string_to_buffer 154
strcvt 155
strlen 156
strncmp, strnicmp 157
strrev 158
strstr 159
substr 160
tolower 161
toupper 162

bdelete

`bdelete` — deletes a number of bytes from a buffer.

Syntax

```
bdelete (buffer, position, length?)
```

Arguments

`buffer`

A buffer.

`position`

The position of the first byte to delete. A number between `0` and the length of the buffer minus `1`.

`length`

An optional number of bytes to delete. The default is `1`. A negative number deletes all bytes to the end. A value of `0` does nothing.

Returns

The number contained at the specified *`position`* in the *`buffer`*, or `nil` if the *`buffer`* is undefined at the given *`position`*.

Description

This function deletes a specified number of bytes from a raw memory buffer. The buffer length does not change as a result of this function. A zero character is placed at the empty position at the end of the buffer, then the buffer is collapsed.

Example

```
Gamma> y = buffer (101, 102, 103, 104);
#{efgh}
Gamma> bdelete(y,1,2);
102
Gamma> y;
#{eh\0h}
Gamma>
```

See Also

`delete`

binsert

`binsert` — inserts a value into a buffer.

Syntax

```
binsert (buffer, position, value)
```

Arguments

`buffer`

A buffer.

`position`

A number giving the zero-based position of the new element within the buffer, or a function.

`value`

A number which will be cast to an 8-bit signed integer.

Returns

The *`value`* inserted.

Description

The binsert function inserts the *`value`* by moving all other values after *`position`* one space to the right, and removing the last value from the buffer.

If *`position`* is a function, it is taken to be a comparison function with two arguments. The value will be inserted using a binary insertion sort with the function as the comparison. A comparison function must return a negative number if the value is ordinally less than the buffer element, 0 if the two are equal, and a positive number if the value is ordinally greater than the buffer element.

Example

```
Gamma> x = string_to_buffer("Hellothere");
#{Hellothere}
Gamma> binsert(x,5,32);
32
Gamma> x;
#{Hello ther}
Gamma>
```

See Also

buffer

buffer

`buffer` — constructs a buffer.

Syntax

`buffer (`*`contents`*`?...)`

Arguments

`contents`

Any Gamma or Lisp expression.

Returns

A buffer containing all of the *`contents`*.

Description

This function constructs a buffer of all of the arguments, in the order they are given.

A buffer is printed as a sequence of characters (some consoles may not support a character for every entry) surrounded by curly brackets and preceded by a hash sign, such as: #{\n+6ALWbe}. This representation of a buffer cannot be read back in to Gamma, so a symbol must be assigned to a buffer in order to refer to or work with it.

Example

```
Gamma> bu = buffer (101, 102, 103, 104, 2 * 25, 4 / 82);
#{efgh2\0}
Gamma> shorten_buffer (bu, 2);
#{ef}
Gamma>
```

See Also

`binsert`, `bdelete`

buffer_to_string

`buffer_to_string` — converts a buffer to a string.

Syntax

```
buffer_to_string (buffer)
```

Arguments

`buffer`

A buffer.

Returns

A string representing the contents of the given *`buffer`* up to the first zero character, or `nil` if the argument is not a buffer.

Description

This function converts the *`buffer`* into a string by treating each element in the buffer as a single character. The first zero character in the buffer terminates the string.

Example

```
Gamma> x = buffer(104,101,108,108,111);
#{hello}
Gamma> buffer_to_string(x);
"hello"
Gamma>
```

See Also

`buffer`

format

`format` — generates a formatted string.

Syntax

```
format (format_string, arguments?...)
```

Arguments

`format_string`

A string containing format directives and literal text.

`arguments`

Expressions which will be matched to format directives on a one-to-one basis.

Returns

A string.

Description

Generates a formatted string using directives similar to those used in the "C" `printf` function. Text in the *`format_string`* will be output literally to the formatted string. Format directives consist of a percent sign (%) followed by one or more characters. The following directives are supported:

- **a** Any Gamma or Lisp expression. The `princ_name` (the same result as applying the string function on the expression) of a Lisp expression is written to the result string.
- **d** An integer number. A numeric expression is cast to a long integer and written to the result string. %d is equivalent to %ld.
- **f** A floating point number. A numeric expression is cast to a long floating point number and written to the result string.
- **g** A floating point number in "natural" notation. A numeric expression is cast to a long floating point number and written to the result string using the most easily read notation which will fit into the given field size, if any. If no field size is specified, use a notation which minimizes the number of characters in the result.
- **s** A character string. A string is written to the result string.

The format directive may contain control flags between the % sign and the format type character. These control flags are:

- **-** Left justify the field within a specified field size.
- **+** Numbers with a positive value will begin with a + sign. Normally only negative numbers are signed.
- **" "** (A space). Signed positive numbers will always start with a space where the sign would normally be.
- **0** A numeric field will be filled with zeros to make the number consume the entire field width.

Format directives may contain field width specifiers which consist of an optional minimal field width as an integer, optionally followed by a period and a precision specified as an integer. The precision has different meanings depending on the type of the field.

- **a** The field width option does not apply to this general case. To specify precision on a s_exp, you can convert it to a string and use the **s** format directives.
- **d** The precision specifies the minimum number of digits to appear. Leading zeros will be used to make the necessary precision.
- **f** The precision specifies the number of digits to be presented after the decimal point. If the precision is zero, the decimal point is not shown.
- **g** The precision specifies the maximum number of significant digits to appear.
- **s** The precision specifies the maximum number of characters to appear.

Example

```
Gamma> pi = 3.1415926535;
3.1415926535000000541
Gamma> format("PI is %6.3f",pi);
"PI is  3.142"
Gamma> alpha = "abcdefghijklmnopqrstuvwxyz";
"abcdefghijklmnopqrstuvwxyz"
Gamma> format("Alphabet starts: %.10s",alpha);
"Alphabet starts: abcdefghij"
Gamma> x = [1,2,3,4,5,6,7,8,9];
[1 2 3 4 5 6 7 8 9]
Gamma> format("x is: %a",x);
"x is: [1 2 3 4 5 6 7 8 9]"
Gamma> format("x is: %.6s",string(x));
"x is: [1 2 3"
Gamma>
```

make_buffer

`make_buffer` — creates a new, empty buffer.

Syntax

```
make_buffer (n_elements)
```

Arguments

`n_elements`

The number of elements (bytes) in the buffer.

Returns

A new buffer.

Description

This function creates a new, empty buffer with *`n_elements`* number of bytes, all set to zero.

Example

```
Gamma> make_buffer(5);
#{\0\0\0\0\0}
Gamma> make_buffer(12);
#{\0\0\0\0\0\0\0\0\0\0\0\0}
Gamma>
```

See Also

`buffer`, `buffer_to_string`

open_string

`open_string` — allows a string to be used as a file.

Syntax

`open_string (`*`string`*`)`

Arguments

`string`

A string.

Returns

A pseudo-file that contains the *`string`* if successful, otherwise `nil`.

Description

This function allows a string to be used as a pseudo-file to facilitate reading and writing to a local buffer. All read and write functions which operate on a file can operate on the result of this call. An attempt to write to the string always appends information destructively to the string. Subsequent reads on the string can retrieve this information. A string is always opened for both read and write.

Example

```
Gamma> s = open_string("Hello there.");
#<File:"String">
Gamma> read_line(s);
"Hello there."
Gamma> s = open_string("Hello there.");
#<File:"String">
Gamma> read(s);
Hello
Gamma> read(s);
there.
Gamma> read(s);
"Unexpected end of file"
Gamma>
```

See Also

`close`, `open`, `read`, `read_char`, `read_double`, `read_float`, `read_line`, `read_long`, `read_short`, `read_until`, `terpri`, `write`, `writec`

parse_string

parse_string — parses an input string.

Syntax

parse_string (*string*, *use_gamma*?=nil, *parse_all*?=nil)

Arguments

string

A character string representing either a Lisp expression or a Gamma statement.

use_gamma

An optional argument that defaults to nil. If nil, the Lisp parser will be used, otherwise the Gamma parser will be used.

parse_all

An optional argument that defaults to nil. If nil, only the first statement in the string will be parsed, otherwise all statements up to the end of the string will be paresed..

Returns

If *parse_all* is nil, return the first statement in the string in internal form. If *parse_all* is non-nil, return all statements in the string as a list of expressions in internal form. If an error occurs during parsing, this function will throw an error.

Description

This function parses the input string using either the Lisp parser or the Gamma parser, and returns either the first complete statement found in the string or all of the statements to the end of the string.

If only the first statement is parsed, the rest of the string is ignored, even if it is invalid. The result is returned in internal form, effectively an executable Lisp representation. Internal form can be passed directly to the eval function for evaluation.

If all statements are returned, they are returned in a list, even if there is only one statement in the string. The resulting list can be passed directly to eval_list.

Example

```
Gamma> a = parse_string("hello");
hello
Gamma> b = parse_string("(cos 5)");
(cos 5)
Gamma> c = parse_string("(+ 5 6) (/ 6 3)");
(+ 5 6)
Gamma> eval(b);
0.28366218546322624627
Gamma> eval(c);
11
Gamma>
```

Using optional arguments:

```
Gamma> parse_string("cos(5);", t);
(cos 5)
Gamma> parse_string("cos(5); sin(5);", t);
(cos 5)
```

```
  Gamma> parse_string("cos(5); sin(5);", t, t);
  ((cos 5) (sin 5))
  Gamma> parse_string ("if (x < 1) y = 1; else y = 0;", t)
  (if (< x 1)
(setq y 1)
(setq y 0)
)
  Gamma>
```

See Also

eval, eval_string, open_string

raw_memory

raw_memory — tells the amount of memory in use.

Syntax

```
raw_memory ()
```

Arguments

none

Returns

The amount of raw memory in use by the system.

Example

```
Gamma> raw_memory();
(72462 818)
Gamma> x = 41;
41
Gamma> raw_memory();
(72787 847)
Gamma> x = 55;
55
Gamma> raw_memory();
(73034 871)
Gamma> y = 10;
10
Gamma> raw_memory();
(73359 900)
Gamma>
```

shell_match

`shell_match` — compares string text to a pattern.

Syntax

```
shell_match (text, pattern)
```

Arguments

text

A text string to compare against the given pattern.

pattern

A shell style pattern.

Returns

`t` if the *text* matches the *pattern*, otherwise `nil`.

Description

This function compares the *text* to the *pattern* using shell-style wildcard rules. The available patterns are as follows:

- ***** matches any number of characters, including zero.
- **[*c*]** matches a single character which is a member of the set contained within the square brackets.
- **[^*c*]** matches any single character which is not a member of the set contained within the square brackets.
- **?** matches a single character.
- **{*xx*,*yy*}** matches either of the simple strings contained within the braces.
- ***c*** (a backslash followed by a character) - matches that character.

Example

To get a directory listing of just *.txt files, use:

```
shell_match(directory("/etc/readme",0,nil),"*.txt");
```

```
Gamma> shell_match("hello","?el[a-m]*");
t
Gamma> shell_match("hello","hel{p,m,ga}");
nil
Gamma>
```

See Also

`apropos`

shorten_buffer

shorten_buffer — reduces the size of a buffer.

Syntax

shorten_buffer (*buffer*, *n_elements*)

Arguments

buffer

The buffer to shorten.

n_elements

The number of elements that the buffer is to be reduced to.

Returns

The shortened buffer.

Description

This function reduces the size of a buffer by cutting off any elements which extend beyond the given size. This function is analogous to the C function, realloc.

Example

```
Gamma> b = buffer(119,120,121,122);
#{wxyz}
Gamma> shorten_buffer(b,3);
#{wxy}
Gamma>
```

See Also

buffer, make_buffer

strchr, strrchr

strchr, strrchr — search a string for an individual character.

Syntax

```
strchr (string, char_as_string)
strrchr (string, char_as_string)
```

Arguments

string

Any character string.

char_as_string

A string containing the one character to be found.

Returns

The position of the *char_as_string* within the *string*, where the first character is in position zero. If the *char_as_string* is not found in the *string*, returns -1. strchr returns the first occurrence of the character in the string. strrchr returns the last occurrence of the character in the string.

Description

These functions search a string for an individual character and return the first or last occurrence of that character within the string. The characters within a string are numbered starting at zero.

Example

```
Gamma> strchr("apple","a");
0
Gamma> strchr("apple","r");
-1
Gamma> strchr("apple","p");
1
Gamma> strrchr("apple pie","p");
6
Gamma>
```

See Also

strcmp, stricmp, string_split, strlen, strrev, strstr, substr

strcmp, stricmp

strcmp, stricmp — compare strings.

Syntax

```
strcmp (string, string)
stricmp (string, string)
```

Arguments

string

Any string.

Returns

A negative number if the first *string* is ordinally less than the second *string* according to the ASCII character set, a positive number if the first *string* is greater than the second, and zero if the two strings are exactly equal.

Description

These functions can be used as comparison functions for sort and insert. stricmp performs the same function as strcmp, but alphabetic characters are compared without regard to case. That is, "A" and "a" are considered equal by stricmp, but different by strcmp.

Example

```
Gamma> strcmp("apple", "peach");
-15
Gamma> strcmp("peach", "apple");
15
Gamma> strcmp("Apple","Apple");
0
Gamma> strcmp("Apple","Apple pie");
-32
Gamma> strcmp("Apple","apple");
-32
Gamma> stricmp("Apple","apple");
0
Gamma>
```

See Also

insert, sort, strchr, strrchr

string

`string` — constructs a string.

Syntax

`string (`*`s_exp`*`...)`

Arguments

`s_exp`

Any Gamma or Lisp expression.

Returns

A string which is the concatenation of all of the arguments.

Description

This function constructs a string by concatenating the `princ` names of all of the arguments. Any argument that can be evaluated will be. No separation is provided between arguments in the resulting string.

Example

```
Gamma> string("A list: ",list(#a,#b,#c), " and a sum: ",2 + 3);
"A list: (a b c) and a sum: 5"
Gamma>
```

See Also

`format`

stringc

`stringc` — constructs a string in Lisp-readable form,

Syntax

`stringc (`*`s_exp`*`...)`

Arguments

s_exp

Any number of expressions.

Returns

A string which is the concatenation of all of the arguments.

Description

This function is identical to the `string` function, except that the result is produced in a form which is guaranteed to be in Lisp-readable form. This means that special characters within strings and symbols will be escaped appropriately for the reader, and that new-line, form-feed, and tab characters are translated into their \n, \f, and \t equivalents.

Example

```
Gamma> string(#my, #symbol);
"mysymbol"
Gamma> stringc(#my, #symbol);
"mysymbol"
Gamma> stringc("A list: ",list(#a,#b,#c), " and a sum: ",2 + 3);
"\"A list: \"(a b c)\" and a sum: \"5"
Gamma>
```

See Also

`string`

string_file_buffer

`string_file_buffer` — queries a string file for its internal buffer.

Syntax

```
string_file_buffer (string_file)
```

Arguments

`string_file`

A file which points to an in-memory string, created by a call to `open_string`.

Returns

The characters remaining to be read within the string file.

Description

This function queries a string file for its internal buffer.

Example

```
Gamma> a = open_string("my false file");
#<File:"String">
Gamma> read_n_chars(a,3);
#{my }
Gamma> string_file_buffer(a);
"false file"
Gamma>
```

See Also

`open_string`

string_split

`string_split` — breaks a string into individual words.

Syntax

`string_split (`*`string`*`, `*`delimiters`*`, `*`max_words`*`)`

Arguments

`string`

Any character string.

`delimiters`

A character string containing delimiter characters.

`max_words`

The maximum number of words to separate.

Returns

A list containing at most (*`max_words`* + 1) elements, each of which is a string.

Description

This function breaks a string into individual words wherever it finds any one of the characters in the *`delimiters`* string. If *`max_words`* is zero or less, there is no limit to the number of words which may be generated. If *`max_words`* is greater than zero, then at most *`max_words`* words will be generated. If there are any characters remaining in the string once *`max_words`* words have been generated, then the remaining characters will be returned as the last element in the result list. If *`delimiters`* is the empty string, "", then the input string will be split at any white space.

Example

```
Gamma> string_split("This is a test","",0);
("This" "is" "a" "test")
Gamma> string_split("This is a test"," ",2);
("This" "is" "a test")
Gamma> string_split("This is a test","ie",0);
("Th" "s " "s a t" "st")
Gamma> string_split("12:05:29",":",-1);
("12" "05" "29")
Gamma>
```

See Also

`strchr`, `strrchr`, `strstr`

string_to_buffer

`string_to_buffer` — creates a buffer object from a string.

Syntax

`string_to_buffer (`*`string`*`)`

Arguments

`string`

The string to be converted to a buffer.

Returns

A buffer whose contents are those of the *`string`*.

Description

This function creates a buffer object from a string. The buffer and string are mapped to different memory areas, so that alterations to one do not affect the other.

Example

```
Gamma> a = "rhino";
"rhino"
Gamma> b = string_to_buffer(a);
#{rhino}
Gamma> a = "hippo";
"hippo"
Gamma> b;
#{rhino}
Gamma>
```

See Also

buffer_to_string

strcvt

`strcvt` — converts the Windows character set of a string.

Syntax

```
strcvt (string, from?, to?)
```

Arguments

`string`

The string that you need to convert.

`from`

An optional argument specifying the Windows code page identifier for the local character set. If no value is entered, the default is `0`, for your system's code page identifier.

`to`

An optional argument specifying the Windows code page identifier of the new character set for the string. If no value is entered, the default is `65001`, for UTF8.

Returns

The converted string.

Description

This function lets Windows users convert the local character set for a given string into a different character set. In many cases, this function is used to convert the local character set into UTF8, and can thus be run with a single *`string`* argument, using the defaults for the *`from`* and *`to`* arguments. A list of valid Windows code page identifiers for various character sets can be found online in the Microsoft documentation, or by searching on the term "`code page identifiers`".

In QNX or Linux this function simply returns the *`string`* argument.

strlen

`strlen` — counts the number of characters in a string.

Syntax

```
strlen (string)
```

Arguments

`string`

A string.

Returns

The number of characters in the *`string`*.

Example

```
Gamma> strlen("Hello");
5
Gamma> strlen("How about a cup of coffee?");
26
Gamma>
```

See Also

`length`

strncmp, strnicmp

strncmp, strnicmp — compare two strings and return a numeric result.

Syntax

```
strncmp (string1, string2, length)
strnicmp (string1, string2, length)
```

Arguments

string1

The first string.

string2

The second string.

length

The maximum length of the comparison.

Returns

An integer < 0 if *string1* is lexically less than *string2* to the given length; 0 if the two strings are equal up to the given length; and an integer > 0 if *string1* is lexically greater than *string2* up to the given length.

Description

The strncmp function compares two strings and returns a numeric result indicating whether the first string is lexically less than, greater than, or equal to the second string. The comparison will carry on for not more than *length* characters of the shorter string. The strnicmp function is the case-insensitive version of strncmp.

Example

```
Gamma> strncmp("hello","helicopter",4);
3
Gamma> strncmp("hello","help",3);
0
Gamma> strncmp("Hello","help", 3);
-32
Gamma> strnicmp("Hello","help", 3);
0
Gamma>
```

See Also

strcmp, stricmp

strrev

`strrev` — reverses the order of characters in a string.

Syntax

`strrev (`*`string`*`)`

Arguments

`string`

A string.

Returns

A new string which is the reverse of the given string.

Description

Automatic, full featured, palindrome creator.

Example

```
Gamma> strrev("I Palindrone I");
"I enordnilaP I"
Gamma> strrev("Madam, I'm adam");
"mada m'I ,madaM"
Gamma> strrev("123456789");
"987654321"
Gamma> strrev("poor dan is in a droop");
"poord a ni si nad roop"
Gamma>
```

See Also

`strchr`, `strrchr`

strstr

`strstr` — finds the location of a given substring.

Syntax

```
strstr (stringA, stringB)
```

Arguments

`stringA`

A string.

`stringB`

A string.

Returns

The position of *`stringB`* within *`stringA`*, or -1 if *`stringA`* does not contain *`stringB`*.

Description

This function finds the first complete occurrence of *`stringB`* within *`stringA`* and returns the position of the starting character of the match within *`stringA`*. The first character in *`stringA`* is numbered zero. If no match is found, -1 is returned.

Example

```
Gamma> strstr("Acme widgets","get");
8
Gamma> strstr("Acme widgets","wide");
-1
Gamma>
```

See Also

`strchr`, `strrchr`

substr

`substr` — returns a substring for a given location.

Syntax

```
substr (string, start_char, length)
```

Arguments

`string`

A string.

`start_char`

The position number of the first character of the substring.

`length`

The length of the substring.

Returns

A new string which is a substring of the input *`string`*.

Description

This function returns a substring of the input string starting at the *`start_char`* position and running for *`length`* characters. The first character in the string is numbered zero. If *`start_char`* is greater than the length of the string, the function returns an empty string. If *`start_char`* is negative, it is indexed from the end of the string. If it is negative and greater than the length of the string, it is treated as zero—the beginning of the string.

If there are fewer characters than *`length`* in the string, or if *`length`* is -1, then the substring contains all characters from *`start_char`* to the end of the string.

Example

```
Gamma> substr("Acme widgets",7,3);
"dge"
Gamma> substr("Acme widgets",9,-1);
"ets"
Gamma> substr("Acme widgets",-7,4);
"widg"
Gamma> substr("Acme widgets",-30,4);
"Acme"
Gamma>
```

See Also

`strchr`, `strrchr`, `string`, `strstr`

tolower

`tolower` — converts upper case letters to lower case.

Syntax

`tolower (`*`string`*`|`*`number`*`)`

Arguments

`string`

Any string.

`number`

Any number.

Returns

Strings with all letters converted to lower case. Numbers in integer form. Floating point numbers are truncated.

Description

This function converts any upper case letters in a string to lower case. It will also convert numbers to their base 10 integer representation.

Example

```
Gamma> tolower("Jack works for IBM.");
"jack works for ibm."
Gamma> tolower("UNICEF received $150.25.");
"unicef received $150.25."
Gamma> tolower(5.3);
5
Gamma> tolower(0b0110);
6
Gamma>
```

See Also

`toupper`

toupper

`toupper` — converts lower case letters to upper case.

Syntax

`toupper (`*`string`*`|`*`number`*`)`

Arguments

`string`

Any string.

`number`

Any number.

Returns

Strings with all letters converted to upper case. Numbers in integer form. Floating point numbers are truncated.

Description

This function converts any lower case letters in a string to upper case. It will also convert numbers to their base 10 integer representation.

Example

```
Gamma> toupper("Jack works for IBM.");
"JACK WORKS FOR IBM."
Gamma> toupper("UNICEF received $150.25.");
"UNICEF RECEIVED $150.25."
Gamma> toupper(5.3);
5
Gamma> toupper(0b0110);
6
Gamma>
```

See Also

tolower

VII. Data Type Conversion

Table of Contents

bin 164
char 165
char_val 166
dec 167
hex 168
int 169
number 170
oct 171
symbol 172

bin

`bin` — converts numbers into binary form.

Syntax

```
bin (number)
```

Arguments

`number`

Any number.

Returns

An integer number in binary format.

Description

This function casts any number to an integer, and returns it in a binary representation. Floating point numbers are truncated.

Example

```
Gamma> bin(12);
0b1100
Gamma> bin(12.9342);
0b1100
Gamma> bin(0x3b);
0b00111011
Gamma> bin(0o436);
0b000100011110
Gamma>
```

See Also

`dec`, `hex`, `oct`

char

`char` — generates an ASCII character from a number.

Syntax

`char (`*`number`*`)`

Arguments

`number`

Any number. This is cast to an integer between 0 and 255. Negative numbers are treated as unsigned 2's complement integers.

Returns

A character string with one character which is the character representation of the ASCII value given as the argument.

Description

This function generates the string representation of an ASCII character value.

Example

```
Gamma> char (65);
"A"
Gamma> char (188);
"¼"
Gamma> char (350.25);
"^"
Gamma> char (-12);
"ô"
Gamma>
```

See Also

`char_val`

char_val

`char_val` — generates a character's numeric value.

Syntax

```
char_val (char_as_string)
```

Arguments

`char_as_string`

A string.

Returns

The ASCII (numeric) value of the first character in the argument string.

Description

Generates the ASCII (numeric) representation of the first character in a string.

Example

```
Gamma> char_val ("A");
65
Gamma> char_val ("q");
113
Gamma> char_val ("hope for all");
104
Gamma> char_val ("3");
51
Gamma> char_val ("ô");
-12
Gamma>
```

See Also

`char`

dec

`dec` — converts numbers into base-10 form.

Syntax

```
dec (number)
```

Arguments

`number`

Any number.

Returns

An integer number in decimal format.

Description

This function casts any number to an integer, and returns it in decimal (base-10) representation.

Example

```
Gamma> dec(0b1100);
12
Gamma> dec(0x3b);
59
Gamma> dec(45.95);
45
Gamma> dec('A');
65
Gamma>
```

See Also

`bin`, `hex`, `oct`

hex

`hex` — converts numbers into hexadecimal form.

Syntax

`hex (`*`number`*`)`

Arguments

`number`

Any number.

Returns

An integer number in hexadecimal format.

Description

This function casts any number to an integer, and returns it in a hexadecimal representation. Floating point numbers are truncated.

Example

```
Gamma> hex (12);
0xc
Gamma> hex (12.9341);
0xc
Gamma> hex (0b111011);
0x3b
Gamma> hex ('r');
0x72
Gamma>
```

See Also

`bin`, `dec`, `oct`

int

`int` — converts to integer form.

Syntax

`int (`*`s_exp`*`)`

Arguments

`s_exp`

Any Gamma or Lisp expression.

Returns

An integer representation of the argument.

Description

This function converts the argument to an integer. Floating point numbers are truncated. Binaries, hexadecimals and characters convert to decimal integers. In strings, if the first character(s) are numerical, they will be converted to an integer. Otherwise, a string will return zero. All other expression types generate zero.

Example

```
Gamma> int(5.5);
5
Gamma> int(0xc);
12
Gamma> int(0b111011);
59
Gamma> int('h');
104
Gamma> int("63 hello");
63
Gamma> int("hello 63");
0
Gamma> int(random());
0
Gamma>
```

See Also

Literals

number

`number` — attempts to convert an expression to a number.

Syntax

`number (`*`s_exp`*`)`

Arguments

`s_exp`

Any Gamma or Lisp expression.

Returns

A numeric representation of the *`s_exp`* if possible, otherwise zero.

Description

The function attempts to convert its argument to a number. Integer and floating point values remain untouched. String arguments are converted to numbers by attempting to read a number from the string starting at the first character in the string. The longest legal number at the beginning of the string is used. All other data types return zero. If possible, the result will be an integer. If the result cannot be represented as an integer, a real (floating point) number is returned.

Example

```
Gamma> number(5);
5
Gamma> number("5.4m");
5.4000000000000003553
Gamma> number("m5.4");
0
Gamma> number(#a);
0
Gamma>
```

oct

`oct` — converts numbers into octal form.

Syntax

```
oct (number)
```

Arguments

`number`

Any number.

Returns

An integer number in octal format.

Description

This function casts any number to an integer, and returns it in an octal representation. Floating point numbers are truncated.

Example

```
Gamma> oct(12);
0o14
Gamma> oct(12.86223);
0o14
Gamma> oct(0x3b);
0o73
Gamma> oct(0b0101101);
0o55
Gamma>
```

See Also

`bin`, `dec`, `hex`

symbol

`symbol` — constructs a symbol from a string.

Syntax

`symbol (`*`string`*`)`

Arguments

`string`

A string.

Returns

A symbol.

Description

This function constructs a symbol whose name is the same as the *`string`*, and places that symbol into the symbol table. Subsequent calls to this function with the same *`string`* will result in the same symbol, preserving the uniqueness of the symbol. Special characters may be included in the symbol name.

Example

```
Gamma> symbol("Strange symbol");
Strange\ symbol
Gamma> Strange\ symbol;
5
Gamma> symbol(string("item",2+3));
item5
Gamma>
```

VIII. Math

Table of Contents

acos, asin, atan, atan2 174
and, not, or 175
band, bnot, bor, bxor 176
ceil 177
cfand, cfor 178
conf, set_conf 179
cos, sin, tan 180
div 181
exp 182
floor 183
log, log10, logn 184
neg 185
pow 186
random 187
round 188
set_random 189
sqr 190
sqrt 191

acos, asin, atan, atan2

acos, asin, atan, atan2 — perform trigonometric arc functions.

Syntax

```
acos (number)
asin (number)
atan (number)
atan2 (number, number)
```

Arguments

number

Any integer or real number. Non-numbers are treated as zero.

Returns

The result of the arc trigonometric function in radians.

Description

These functions perform the arc trigonometric functions arc cosine, arc sine, arc tangent, and arc tangent with 2 arguments. The atan2 function is equivalent to:

```
atan( y / x );
```

except that atan2 is able to correctly handle x and y values of zero.

Example

```
Gamma> acos (0.5);
1.0471975511965978534
Gamma> asin (0.5);
0.52359877559829892668
Gamma> atan (2);
1.107148717794090409
Gamma> atan2 (1, 2);
0.46364760900080609352
Gamma> atan2 (1, 0);
1.570796326794896558
Gamma> atan2 (0, 2);
0
Gamma>
```

See Also

sin, cos, tan

and, not, or

`and, not, or` — are the same as the corresponding Logical Operators.

Syntax

```
and (!condition[,!condition]...)
not (condition)
or (!condition[,!condition]...)
```

Arguments

condition

Any Gamma or Lisp expression.

Returns

Non-`nil` or `nil`.

Examples

```
Gamma> not(6);
nil
Gamma> not(nil);
t

Gamma> and(5<6,string("hi ","there"));
"hi there"
Gamma> and(5>6,string("hi ","there"));
nil

Gamma> x = 5;
5
Gamma> y = 6;
6
Gamma> or(x == 3, y == 0);
nil
Gamma> or(x == 3, y == 6);
t
Gamma>
```

See Also

Logical Operators

band, bnot, bor, bxor

band, bnot, bor, bxor — perform bitwise operations.

Syntax

```
band (number, number)
bnot (number)
bor (number, number)
bxor (number, number)
```

Arguments

number

Any number. Non-numbers are treated as zero.

Returns

An integer which is the result of the particular operation.

Description

The binary operations cast their arguments to integers, and then perform bitwise operations to produce an integer result.

- **band** bitwise AND
- **bnot** bitwise NOT (inversion of all bits in a 32-bit word)
- **bor** bitwise OR
- **bxor** bitwise exclusive OR (XOR)

Example

```
Gamma> band(7,5);
5
Gamma> bnot(7);
-8
Gamma> bor(7,5);
7
Gamma> bxor(7,5);
2
Gamma>
```

See Also

Bitwise Operators

ceil

`ceil` — rounds a real number up to the next integer.

Syntax

```
ceil (number)
```

Arguments

`number`

Any number. Non-numbers are treated as zero.

Returns

The smallest integer that is greater than or equal to the *`number`*.

Description

This function has the effect of rounding real numbers up to the next integer. Integers are unaffected.

Example

```
Gamma> ceil(1.1);
2
Gamma> ceil(-1.1);
-1
Gamma> ceil(4);
4
Gamma>
```

See Also

`floor`, `round`

cfand, cfor

cfand, cfor — perform and and or functions with confidence factors.

Syntax

```
cfand (!s_exp[,!s_exp]...)
cfor (!s_exp[,!s_exp]...)
```

Arguments

condition

Any Gamma or Lisp expression.

Returns

A confidence factor, an integer between 0 and 100.

Description

These functions determine the confidence factor of one or more expressions. cfand returns the lowest confidence factor among all of the passed *conditions*, while cfor returns the highest confidence factor among the *conditions*.

Example

```
Gamma> a = 3;
3
Gamma> b = 4;
4
Gamma> set_conf(a,50);
50
Gamma> set_conf(b,10);
10
Gamma> cfand(a,b);
10
Gamma> cfor(a,b);
50
Gamma>
```

See Also

conf

conf, set_conf

conf, set_conf — query and set confidence factors.

Syntax

```
conf (s_exp)
set_conf (s_exp, number|s_exp)
```

Arguments

s_exp

Any Gamma or Lisp expression.

number|*s_exp*

Any number, or any expression that evaluates to a number. Non-numbers are treated as zero.

Returns

The confidence factor of the *number* or *s_exp*.

Description

All Gamma and Lisp expressions in Gamma have an associated confidence factor between 0 and 100 which may be queried using the conf function. This is typically 100, or fully confident. Exceptions arise only when the user explicitly sets the confidence to another value, or when the DataHub provides a confidence value to the interpreter. The set_conf function will set the confidence of an expression to any numerical value, though legal values are between 0 and 100. Numbers less than 0 indicate indeterminate confidence. Numbers greater than 100 will produce strange results.

Example

```
Gamma> x = 3;
3
Gamma> set_conf(x, 40);
40
Gamma> conf(x);
40
Gamma>
```

cos, sin, tan

`cos, sin, tan` — perform trigonometric functions.

Syntax

```
cos (number)
sin (number)
tan (number)
```

Arguments

number

Any number in radians. Non-numbers are treated as zero.

Returns

The result of the trigonometric functions cosine, sine and tangent.

Example

```
Gamma> cos(8);
-0.14550003380861353808
Gamma> sin(.8);
0.71735609089952279138
Gamma> tan(.5);
0.54630248984379048416
Gamma>
```

See Also

asin, acos, atan, atan2

div

`div` — divides, giving an integer result.

Syntax

```
div (number, number)
```

Arguments

`number`

Any number.

Returns

The integer result of the division of the first argument by the second.

Description

This function is equivalent to (`floor` (number/number)) .

Example

```
Gamma> div(12,5);
2
Gamma> div(23423,899);
26
Gamma>
```

See Also

Arithmetic Operators

exp

`exp` — calculates an exponent of the logarithmic base (e).

Syntax

`exp (`*`number`*`)`

Arguments

`number`

Any number.

Returns

The natural logarithmic base, e, raised to the power of the *`number`*.

Example

```
Gamma> exp(0);
1
Gamma> exp(3);
20.085536923187667924
Gamma>
```

floor

`floor` — rounds a real number down to its integer value.

Syntax

```
floor (number)
```

Arguments

`number`

Any number. Non-numbers are treated as zero.

Returns

The largest integer which is less than or equal to the *`number`*.

Example

```
Gamma> floor(1.2);
1
Gamma> floor(1.9);
1
Gamma> floor(-1.2);
-2
Gamma> floor(-1.9);
-2
Gamma>
```

See Also

`ceil`, `round`

log, log10, logn

log, log10, logn — calculate logarithms.

Syntax

```
log (number)
log10 (number)
logn (base, number)
```

Arguments

number

Any numeric value.

base

The logarithmic base.

Returns

For log, the natural logarithm of the argument. For log10, the base 10 logarithm of the argument. For logn, the logarithm of the number in the given base.

Description

Non-numeric arguments are treated as zero. Illegal values for the arguments will cause an error.

Example

```
Gamma> log(2);
0.69314718055994528623
Gamma> log10(2);
0.30102999566398119802
Gamma> logn(8,2);
2.999999999999995559
Gamma>
```

neg

`neg` — negates.

Syntax

```
neg (number)
```

Arguments

`number`

Any number.

Returns

The negative of the *`number`*.

Example

```
Gamma> neg(5);
-5
Gamma> neg(-5);
5
Gamma>
```

pow

`pow` — raises a base to the power of an exponent.

Syntax

```
pow (base, exponent)
```

Arguments

`base`

Any number.

`exponent`

Any number.

Returns

The result of raising the *`base`* to the given *`exponent`*.

Description

Calculates a base to the power of an exponent. Non-numbers are treated as zero.

Example

```
Gamma> pow(2,3);
8
Gamma> pow(12,2);
144
Gamma> pow(5.2,4.75);
2517.7690015606849556
Gamma>
```

random

`random` — generates random numbers from 0 to 1.

Syntax

```
random ()
```

Arguments

none

Returns

A floating point random number which is greater than or equal to 0 and is less than 1.

Description

This function uses a pseudo-random number generator to generate a non-repeating sequence of numbers randomly distributed across the range of 0 <= x < 1.

The random number generator should be seeded prior to being called by using the `set_random` function. If the same seed is given to `set_random`, the same random sequence will result every time.

Example

```
#!/usr/local/bin/gamma -d

//Seed random number generator to clock setting:
set_random(clock());

//Randomly generate an integer from one to six:
function one_to_six ()
{
        floor(6 * random()) + 1;
}

//Princ the results:
x = one_to_six();
princ(x, "\n");
```

See Also

`set_random`

round

`round` — rounds a real number up or down to the nearest integer.

Syntax

```
round (number)
```

Arguments

`number`

A number.

Returns

The nearest integer to the *`number`*.

Description

This function rounds its argument to the nearest integer. Values of .5 are rounded up to the next highest integer.

Example

```
Gamma> round(8.73);
9
Gamma> round(2.21);
2
Gamma> round(5.5);
6
Gamma> round(5.49);
5
Gamma>
```

See Also

`ceil`, `floor`

set_random

set_random — starts random at a different initial number.

Syntax

```
set_random (integer_seed)
```

Arguments

integer_seed

Any integer number.

Returns

t

Description

This function seeds the random number generator to start the pseudo-random sequence at a different number. The same *integer_seed* will always produce the same pseudo-random sequence. set_random is commonly called with an unpredictable *integer_seed*, such as the result of clock.

Example

```
Gamma> set_random(95);
t
Gamma> random();
0.26711518364027142525
Gamma> random();
0.8748339582234621048
Gamma> random();
0.30958001874387264252
Gamma> set_random(clock());
t
Gamma> random();
0.41952831624075770378
Gamma> random();
0.99278739839792251587
Gamma> random();
0.42997436970472335815
Gamma>
```

See Also

random

sqr

`sqr` — finds the square of a number.

Syntax

```
sqr (number)
```

Arguments

`number`

Any number.

Returns

The square of the *`number`*.

Example

```
Gamma> sqr(11);
121
Gamma> sqr(32.73);
1071.2528999999997268
Gamma>
```

See Also

`sqrt`

sqrt

`sqrt` — finds the square root of a number.

Syntax

```
sqrt (number)
```

Arguments

`number`

Any number.

Returns

The square root of the *`number`*.

Example

```
Gamma> sqrt(9);
3
Gamma> sqrt(144);
12
Gamma> sqrt(95);
9.7467943448089631175
Gamma>
```

See Also

sqr

Index

Symbols

!, 32
!=, 29
#, 33
$, 34
%, 21
%=, 23
&, 25
&&, 32
&=, 23
’, 33
(, 30
(), 20
), 30
*, 21
*=, 23
+, 21
++, 31
+++, 31
+=, 23
„ 30, 33
-, 21
--, 31
---, 31
-=, 23
., 27
.., 27
/, 21
/=, 23
::=, 22
:=, 22
<, 29
<<, 25
<<=, 23
<=, 29
=, 22
==, 29
>, 29
>=, 29
>>, 25
>>=, 23
? :, 35
@, 33
\, 34
^, 25
^=, 23
_all_tasks_, 11
_atexit_functions_, 11
_auto_load_alist_, 11
_case_sensitive_, 11
commasplice, 11
comma, 11
_current_input_, 11
debug, 11
eof, 11
eol, 11
_error_stack_, 11
_eval_silently_, 11
_eval_stack_, 11
event, 11
_fixed_point_, 11
gui, 11
_gui_version_, 11
_ipc_file_, 11
_jump_stack_, 11
_last_error_, 11
_load_extensions_, 11, 95
_os_, 11
_os_release_, 11
_os_version_, 11
_require_path_, 11, 95
timers, 11
undefined, 11
_unwind_stack_, 11
‘, 33, 94
|, 25
||, 32
~, 25

A

acos, 174
alist_p, 5
and, 175
append, 104
aref, 105
array, 106
array_p, 5
array_to_list, 107
aset, 108
asin, 174
assoc, 109
assoc_equal, 109
atan, 174
atan2, 174
autotrace_p, 5

B

backquote, 33, 94
band, 176
bdelete, 135
bin, 164
binsert, 136
bnot, 176
bor, 176
breakpoint_p, 5
bsearch, 110
buffer, 137
buffer_p, 5
buffer_to_string, 138
builtin_p, 5
bxor, 176

C

caaar, 111
caadr, 111
caar, 111
cadar, 111
caddr, 111
cadr, 111
call, 60
car, 111
cdaar, 111
cdadr, 111
cdar, 111
cddar, 111
cdddr, 111
cddr, 111
cdr, 111
ceil, 177
cfand, 178
cfor, 178
char, 165
char_val, 166
class, 14, 37
class_add_cvar, 61
class_add_ivar, 62
class_name, 63
class_of, 64
class_p, 5
collect, 14
condition, 39
conf, 179
cons, 112
constant_p, 5
cons_p, 5
copy, 113
copy_tree, 114
cos, 180

D

dec, 167
defclass, 65
defmacro, 66
defmacroe, 66
defmethod, 68
defun, 67
defune, 67
defvar, 69
delete, 115
destroy, 70
destroyed_p, 5
difference, 116
div, 181
do, 14

E

else, 14
eq, 71
equal, 71
error, 73
eval, 74
eval_list, 75
eval_string, 76
exp, 182

F

file_p, 5
find, 117
find_equal, 117
fixed_point_p, 5
floor, 183
for, 14, 40
force, 77
forceq, 77
forceqq, 77
format, 139
funcall, 78
function, 14, 41
function_args, 79
function_body, 80
function_name, 81
function_p, 5

G

gamma, 17
getprop, 82

H

has_cvar, 83
has_ivar, 84
hex, 168

I

if, 14, 43
insert, 118
instance, 90
instance_p, 5
instance_vars, 85
int, 169
intersection, 119
int_p, 5
is_class_member, 86
ivar_type, 87

L

length, 120
list, 121
listq, 121
list_p, 5
list_to_array, 122
load, 95
load_lisp, 95
local, 14, 45
log, 184
log10, 184
logn, 184
long_p, 5

M

macro, 88
macro_p, 5
make_array, 123
make_buffer, 141
method, 14, 47
method_p, 5

N

nappend, 124
neg, 185
new, 90
nil, 16
nil_p, 5
noeval , !, 11
not, 175
nremove, 125
nreplace, 126
nreplace_equal, 126
nth_car, 127
nth_cdr, 127
number, 170
number_p, 5

O

oct, 171
open_string, 142
optional , ?, 11
or, 175

P

parent_class, 91
parse_string, 143
phgamma, 17
pow, 186
predicate, 5
print_stack, 92
prog1, 49
progn, 49
properties, 93
protect unwind, 50

Q

quote, 94

R

random, 187
raw_memory, 145
real_p, 5
registered_p, 5
remove
  Gamma, 128
require, 95
required_file, 95
require_lisp, 95
rest , ..., 11

reverse, 129
round, 188
rplaca, 130
rplacd, 130

S

set, 97
setprop, 98
setprops, 99
setq, 97
setqq, 97
set_conf, 179
set_random, 189
shell_match, 146
shorten_array, 131
shorten_buffer, 147
SIGABRT, 11
SIGBUS, 11
SIGCHLD, 11
SIGCONT, 11
SIGDEV, 11
SIGEMT, 11
SIGFPE, 11
SIGHUP, 11
SIGILL, 11
SIGINT, 11
SIGIO, 11
SIGKILL, 11
SIGNAL_HANDLERS, 11
SIGPIPE, 11
SIGPOLL, 11
SIGPWR, 11
SIGQUIT, 11
SIGSEGV, 11
SIGSTOP, 11
SIGSTST, 11
SIGSYS, 11
SIGTERM, 11
SIGTTIN, 11
SIGTTOU, 11
SIGURG, 11
SIGUSR1, 11
SIGUSR2, 11
SIGWINCH, 11
sin, 180
sort, 132
sqr, 190
sqrt, 191
strchr, 148
strcmp, 149
strcvt, 155
stricmp, 149
string, 150
stringc, 151
string_file_buffer, 152
string_p, 5
string_split, 153
string_to_buffer, 154
strlen, 156
strncmp, 157
strnicmp, 157
strrchr, 148
strrev, 158
strstr, 159
substr, 160
switch, 51
symbol, 172
symbol_p, 5
sym_alist_p, 5

T

t, 15
tan, 180
tolower, 161
toupper, 162
trap_error, 100
true_p, 5
try catch, 53
type, 5

U

undefined_p, 5, 8
undefined_symbol_p, 5, 8
union, 133
unwind_protect, 101

W

whence, 102
while, 14, 55
with, 14, 56

Colophon

This book was produced by Cogent Real-Time Systems, Inc. from a single-source group of SGML files. Gnu Emacs was used to edit the SGML files. The DocBook DTD and related DSSSL stylesheets were used to transform the SGML source into HTML, PDF, and QNX Helpviewer output formats. This processing was accomplished with the help of OpenJade, JadeTeX, Tex, and various scripts and makefiles. Details of the process are described in our book: Preparing Cogent Documentation, which is published on-line at
`http://developers.cogentrts.com/cogent/prepdoc/book1.html`.

Text written by Andrew Thomas, Mark Oliver, Bob McIlvride, and Elena Devdariani.