Documentation Library

Gamma™ Programmer's Manual

Version 7.2

Cogent Real-Time Systems, Inc.

August 15, 2012

Gamma™ Programmer's Manual: 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

List of Tables

Chapter 1. Introduction

1.1. 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.

1.2. Assumptions about the Reader

This guide assumes you are familiar with at least one programming language.

The syntax of Gamma is very similar to C, so programmers familiar with C can start programming in Gamma almost immediately. Knowledge of pointers and memory allocation is not necessary for using Gamma.

1.3. 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.

1.4. Download and Installation

You can download Gamma from the Cogent Web Site, and then follow these instructions for installing it on your system.

Cogent software comes packaged in self-installing archives available for download, or on diskette for commerically-licensed packages. Each software package name, which we refer to in these instructions as *`software_package_name`*, contains the product name, version number, operating system and sometimes other information, and will end with either `.sh.gz` or `.qpr`. For example, `gamma-4.0-bin-48-Linux.sh.gz` or `CascDataHub-4.0-bld10-x86-Cogent.qpr` are typical package names. The installation procedure is standardized across Cogent products, but depends on the operating system.

1.4.1. QNX 4

Option A: Install the archive from diskette.

1. Log in as root.
2. Insert the program diskette into your QNX 4 computer.
3. Type the command: **`install`**

   and respond to the system prompts.

Option B: Install the archive from a download or received as an e-file.

1. Download or copy the *`software_package_name`*`.sh.gz` file onto your QNX 4 computer.
2. Log in as root.
3. Type the command: **`gunzip` *`software_package_name`*`.sh.gz`**

   This unzips the software package, and removes the `.gz` extension from the end of the filename.
4. Type the command: **`sh` *`software_package_name`*`.sh`**

   and respond to the system prompts.

If you get an error trying to install the `.sh` archive in QNX, please read the Installing program archives in QNX section of the Glossary, FAQ and Troubleshooting for help.

1.4.2. QNX 6

Option A: Use the QNX 6 Installer program. The Cogent repository is located at http://developers.cogentrts.com/repository.

Option B: Download the *software_package_name*.qpr file using the QNX 6 Voyager browser. The archive will install automatically.

Option C: Download or copy from diskette the *software_package_name*.qpr file onto your QNX 6 computer. Then (as root) run the command:

```
qnxinstall software_package_name.qpr
```

and respond to the system prompts.

1.4.3. Linux

First make sure the SRR kernel module is installed. If not, it is downloadable from the SRR for Linux page of the Cogent web site. Then follow these instructions to install the software package:

1. Download or copy from diskette the *software_package_name*.sh.gz file onto your Linux computer.
2. Log in as root.
3. Type the command: **gunzip *software_package_name*.sh.gz**

   This unzips the software package, and removes the .gz extension from the end of the filename.
4. Type the command: **sh *software_package_name*.sh**

   and respond to the system prompts.

1.4.4. Installed file locations

On whichever OS the software is installed, all files will be written to the /usr/cogent/ directory. Depending on which packages are installed, the following subdirectories will contain the types of files shown:

bin/	Binary executables.
dll/	Dynamically-linked libraries.
docs/	Miscellaneous documentation. (Regular documentation is downloaded separately.)
include/	Header files.
lib/	Cogent library files.
license	The license file (see below).
require/	Lisp or Gamma files used by Gamma or its extensions.
src/	The source code for examples, tests, tutorials, etc.

1.4.5. Installing licenses

Licenses to use the software can be purchased from Cogent. To install a license, you need to copy the license string into the /usr/cogent/license file. If this file does not exist on your system, just create one as a text file and list the license strings, one per line.

If a license is not installed, you will see the following console message on startup:

```
software_package_name: No valid licenses found.
This program is running in demo mode and will terminate after 1 hour.
```

and the software will run for one hour in demo mode.

1.5. Cogent Product Integration

Cogent products work together to support real-time data connectivity in Windows, Linux, and QNX. They can be dynamically integrated as a group of modules where each module connects to any other module(s) as needed. New modules can be added and existing modules reconfigured or modified, all during run-time. Data in any module of the system can be collected and redistributed to any other module via the Cascade DataHub and Cascade Connect. Communication with field devices is provided by one of several Cogent Device Drivers. Historical records of unlimited size can be maintained and queried with the Cascade Historian, and ASCII text files can be logged with the Cascade TextLogger.

Custom programs written in C or C++ can interface with the system, using the Cogent C API or the DataHub APIs for C++, Java, and .NET. In addition, Cogent's own dynamically-typed object-oriented programming language, Gamma, is fully compatible with all modules. User interfaces can be created in Gamma, which supports Photon in QNX and GTK in Linux.

1.6. Where can I get help?

If you are having problems with a Cogent product, first check the Troubleshooting Guide. If you can't find the answer there, you can contact Cogent Real-Time Systems, Inc. for technical support for any product you have purchased.

- Email: <`support@cogent.ca`>
- Phone: 1-888-628-2028
- Fax: (905) 702-7850

Chapter 2. Getting Started

2.1. Interactive Mode

You can invoke the Gamma engine by typing

```
[sh]$  gamma
```

at the shell prompt. (If you are using Gamma with Photon, you should type **`phgamma`**.)

It will return the following Gamma prompt:

```
Gamma>
```

Now you can start writing instructions to Gamma and get an immediate response. If you define a variable `a` without assigning it a value, Gamma will respond with the message that the symbol is undefined and suggest debugging:

```
Gamma> a;
Symbol is undefined: a
debug 1>
```

Type **`Ctrl - D`** to return to the Gamma prompt and assign `a` a value:

```
Gamma> a = 5;
5
Gamma>
```

This time Gamma responds with the value assigned to the variable. In Gamma a variable must be assigned a value. The library function `undefined_p` can be used to test if a variable is defined:

```
Gamma> undefined_p (b);
t
Gamma> b = 1;
1
Gamma> undefined_p (b);
nil
Gamma>
```

This function returns `t` for true and `nil` for false. The objects *t* and *nil* are discussed in more detail in the Logical Types section of the Basic Data Types and Mechanisms chapter.

A function is defined in Gamma using a *function* statement:

```
Gamma> function MyFunc (a) { a *= 10;}
(defun MyFunc (a) (*= a 10))
Gamma>
```

Gamma returns the function definition in Lisp syntax. It reflects the fact that Gamma is using the Lisp engine internally. Basically, Lisp displays functions as lists, surrounded by parentheses. The first word in every Gamma function definition is `defun` because that is the Lisp function for defining functions. After that, the function name is listed, followed by its arguments and code, which is also in Lisp format.

The `MyFunc` function called with 12 as its argument will return the value 120, as follows:

```
Gamma> MyFunc (12);
120
Gamma>
```

Notice that no type specification is used. An important feature of Gamma is that it is an abstractly typed language, making it unnecessary to specify the type of a data object in order to be able to use it. This does not mean that objects do not have types, but rather the system does not require that the type of an object be known until the code actually executes. This allows a function to return entirely different types, depending on what the calculation produces.

For example, a `MIN` function could be defined as:

```
function min (x, y) { if (x < y) x; else y; }
```

This function will return an integer or floating point number depending on the types of the arguments. The arguments do not need to be the same type. Gamma automatically type-casts them, favoring the smallest possible memory allocation. However, the less-than comparison will fail if both arguments are not numeric types.

Finally, `return` statements are not necessary in Gamma. Looking at the function MyFunc, we see it returns 120, the result of multiplying `a` by 10. Any Gamma function that executes successfully always returns a value, which is the result of evaluating the last expression in the function. This return value determines the value and type of the function.

This topic is discussed in greater detail in Function Definitions in the Functions and Program Structure chapter.

2.2. Executable Programs

There is currently no facility for embedding Gamma source code into a stand-alone Gamma executable. However, it is possible to create a Gamma program which appears to be a stand-alone executable to the user. This is done by using the shell's **#!** directive at the top of the Gamma file. Simply include the line indicating the full path for your Gamma program, for example:

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

If using Gamma with Photon, you should use:

```
#!/usr/cogent/bin/phgamma
```

Then change the permissions on the Gamma file to be executable. The Gamma file can now be run directly as if it were a stand-alone executable. The directive must appear as the first line of the program, and it must reference the actual Gamma executable, not a link.

The following example program will print the famous "Hello world" message to the screen:

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

princ ("Hello world\n");
```

There is no requirement for a function named `main`. If a function with the name `main` is defined, then Gamma automatically starts to execute that function. Thus, the result of the following program:

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

function MyPrint()
{
    princ ("Hello world\n");
}

function main()
{
    MyPrint();
```

```
}

MyPrint();
```

is that the function `MyPrint` will be executed twice.

Command line arguments can be passed to the Gamma program using the list variable `argv`. For more details see Tutorial I.

2.3. Symbols and Values

A *symbol* is a fundamental notion in Gamma. It is a unique word or name within the program. In this sense all symbols are global in Gamma (although they can have local scope), and all references to a particular symbol will be the exact same Gamma object.

A symbol can contain: lowercase letters, uppercase letters, digits, and underscores. Using a symbol character operator, other characters can be used. A '\' character escapes the next character. A '$' escapes any and all of the characters in the symbol.

Symbols can have values so they can be used as variables. For the time being it is probably easier to think of a symbol as a variable. (For other uses of symbols see the Advanced Types and Mechanisms chapter of this guide). The same symbol can have different values depending on the part of the program in which the reference to the symbol is made.

For example:

```
Gamma> i = 5;
5
Gamma> function MyLocal (n) { local i; for (i = 1; i < n; i++) princ("i = ", i, "\n");}
(defun ...)
Gamma> MyLocal (3);
i = 1
i = 2
2
Gamma> i;
5
```

In this example the symbol `i` is created and the value of `5` is assigned to it. Then in the definition of function `MyLocal` the value of `i` is declared to be *local* to the function (or to have *local scope*). The function prints the local values of `i` and returns the result of the last calculation. After `MyLocal` returns, the value of the symbol `i` remains `5`, as defined at the beginning.

Unlike many procedural languages such as C and Pascal, Gamma uses *dynamic scoping*. That means that if a function defines a local variable, then that variable becomes global to any functions it calls. In the example above, suppose we define `MyScope` in the higher call, or at the global level:

```
Gamma>  function MyScope (s) {........}
```

Then we modify the function `MyLocal` such that now it calls `MyScope`:

```
Gamma>  function MyLocalNew (n)
{
  local i;
  for (i = 1; i < n; i++)
    {
      princ("i = ", i, "\n");
      MyScope(i);
    }
}
```

The value of `i` used as an argument for `MyScope` will be the value of the local variable most recently declared in the calling sequence: `MyScope (1)`, `MyScope (2)`, etc. The scope of the variable is thus determined at run time by the order in which functions are called.

Dynamic scoping in Gamma is very different from the convention in C known as lexical scoping, where the scope of a variable is determined according to the syntactic position in a program where it is declared.

Chapter 3. Basic Data Types and Mechanisms

3.1. Numeric Types

A number in Gamma is any integer, real, or fixed-point value.

3.1.1. Integer

An integer is any 32-bit integer number. Integers can be read and written in the following representations:

1. decimal

   Example: 45, 129000

2. hexadecimal (start with 0x)

   Example: 0xf12

3. octal (start with 0o)

   Example: 0o777

4. binary (start with 0b)

   Example: 0b101001

5. character (enclosed by ’ ’)

   Example: ’s’

3.1.2. Real

A real number is a 64-bit double precision floating point number. It can contain a decimal point and it may end with the letter `e` followed by a signed exponent.

Examples: 0.1, 235.02013576, 5e-2, 3.74e-7

3.1.3. Fixed-point Real

A fixed-point real number is one that is represented by an integer, where the high 16 bits represent an integer value, and the low 16 bits represent a mantissa. There are very few reasons to work with fixed-point numbers unless floating-point error in numeric comparison is intolerable. Fixed-point numbers are created by any numeric function so long as the value of the symbol `_fixed_point_` is non-`nil`. The default value of `_fixed_point_` is `nil`, so that floating point numbers of type real are created by default.

3.1.4. Number Operators

The following operators can be used with numeric data:

- arithmetic (+, -, *, /, %, div)

  Example:

  ```
  Gamma> 2 + 3;
  5
  ```

- logical (!, &&, ||)

  Example:

  ```
  Gamma> !2.75;
  nil
  ```

- comparison (!=, ==, <, <=, >, >=)

  Example:

  ```
  Gamma> 2 != 3;
  t
  ```

- bitwise (<<, >>, ~, &, |, ^)

  Example:

  ```
  Gamma> bin(25);
  0b00011001
  Gamma> bin(25 << 1);
  0b00110010
  ```

3.2. Logical Types

In Gamma, there are two objects of logical type: *t* and *nil*. `t` is a logically true value, and `nil` is the ONLY logically false value in Gamma. All other objects are considered to be logically true, including the number zero. This is different from the C language where the number zero is treated as logically false.

The operators `!`, `&&`, and `||` can be used with `t` and `nil`.

Example:

```
Gamma> !nil;
t
Gamma> !t;
nil
```

```
Gamma> !0;
nil
Gamma> 2 || nil;
2
Gamma> 2 && nil;
nil
```

3.3. Strings

A string is a sequence of characters (whose values range from 0x01 to 0xff), stored in consecutive bytes of memory, terminated by the null character \0. Unlike symbols, strings are not unique within the system. Strings are denoted by enclosing them in double quotation marks (for example: `"A string."`). A string is created by any of the string functions (particularly `string`), or by reading a string constant in the form **`"A string."`**.

The following types of operators can be used with strings:

- comparison (!=, ==)

  Example:

  ```
  Gamma> "cat" != "dog";
  t
  Gamma> "cat" == "dog";
  nil
  ```

- logical (!, &&, ||). Strings have the logical value of true in Gamma.

  Example:

  ```
  Gamma> "cat" && "dog" && nil;
  nil
  Gamma> "cat" || "dog" || nil;
  "cat"
  ```

However, strings are normally manipulated in Gamma using the string functions (see Strings and Buffers in the Reference Manual.)

3.4. Lists and Arrays

An *array* is an ordered collection of elements, each of which can be of a different type. Each element is associated with a fixed numeric index into the array, and can be set or read using the functions `aset` and `aref`, or through the use of square brackets [ ] . If an attempt is made to set an array at an index beyond the end of the array, the array will increase in size to the given index, filling any undefined intermediate values with `nil`. If an attempt is made to read an element from an array beyond the size of the array, then `aref` will return `nil`, and no error will be reported.

An array is created by a call to `array`, or by reading a Lisp expression of the form `[element element ...]` . A side effect of creating an array using the `[...]` syntax is that the array will be

effectively static, in the sense that if the array is defined within a function, then it will not be re-created during each call to the function. It would appear to exist within the code space of the function.

A *list* is a linked group of consecutive elements called cons cells. A cons cell consists of a reference to the list element and a forward pointer to the next cons cell. This organization necessarily means that a list is single-directional, and also means that any cons cell within the list is the head of another list, consisting of that cons cell and all cells after it in the list. The last cons cell in a list normally has a forward pointer of `nil`.

It is possible to create a list whose last cons cell points to a non-`nil` object. For example, a single cons cell could have a data value of `1`, and a forward pointer to number `2`. This would create a *dotted* list, written as `(1 . 2)` (note the spaces around the dot). Only the last element in a list can be a dotted cons cell. Thus, it is not possible to create a list `(1 . 2 3)` or `(1 . (2 3))`. A dotted cons cell is created by a call to `cons`, or by reading a Lisp expression of the form `(x . y)`. The form `(x . nil)` is equivalent to `(x)`.

There are two very important functions used to access the members of a list: `car(list)` and `cdr(list)`. The `car` function returns the first element of a list. For example, `car(list(3,2,1))` returns 3. The `cdr` function returns the "tail" of the list, that is, the list without the first element. For example, `cdr(list(3,2,1))` returns (2 1). The combinations of these two functions allow access to any element(s) of a list. For example, `car(cdr(list(3,2,1)))` returns 2, the second element of the list. Normally a shortcut `cadr(list)` is used for `car(cdr(list))`.

For more information see `car, cdr` in the Reference Manual.

A list is created by a call to `cons` or `list`, or by reading a Lisp expression of the form `(x ...)`, or by evaluating a Lisp expression of the form `'(x ...)` or `‘(x ...)`.

Both lists and arrays can be traversed using the `with` statement. For example:

```
Gamma> sum = 0;
0
Gamma> with i in list(1,2,3) do {sum += i;}
nil
Gamma> sum;
6
Gamma>
```

For more information on the `with` statement, see `with` in the Reference Manual.

3.5. Constants

A constant is any symbol that has been defined as such with the assignment operator `::=`. Note that since the check is made at run-time, the constant is protected, even if the symbol name is evaluated at run-time.

Example:

```
Gamma> e ::= 2.17128128;
2.17128128
Gamma> e = 3;
Assignment to constant symbol: e
debug 1>

(Type Ctrl - D to return to
the Gamma prompt.)

Gamma>
```

3.6. Operators and Expressions

Gamma provides operators for the basic arithmetic calculations: addition, subtraction, multiplication, division and taking the modulus. There is also a group of assignment, bitwise, comparison, increment and decrement, logical, and quote operators. For information on operator precedence and associativity see Operators in the Reference Manual.

Operators are used with one, two, or three values to create *expressions*. For instance, `4 + 2` is an expression whose value is `6. In` Gamma, every expression has a value.

Not all expressions contain operators, though. The number `5`, for example, is also an expression. Generally speaking, an expression in Gamma is anything that can be evaluated. This includes numbers, symbols that have been assigned values, strings, `t`, `nil`, constants, lists, arrays, and so on.

These are also known as *symbolic* expressions, a term that has been abbreviated to *s_exp*. Since expressions are often used as arguments for functions, you will come across the parameter *`s_exp`* in function definitions in the Reference Manual.

An expression can become a *statement* by adding a semicolon. Thus, `4 + 2;` is a statement. For more information on statements, see the Statements section in the Control Flow chapter.

3.7. Comments

There are two ways insert comments into Gamma code. To comment out a line, you can use a double slash (`//`) like this:

```
a = 5;
b = 7;

// This assigns the value of c.

c = a + b;
```

To comment out a block of text, you can put the symbol `/*` at the beginning, and `*/ at the end, like this:`

```
a = 5;
b = 7;

/* This assigns the value of c,
   which is very important to the
   future of our project.*/

c = a + b;
```

It is not permitted to put an unmatched double quote mark (`"`) into the second type of comment. This is a feature that allows you to comment out a string, to include a comment mark in a string, and even to comment out a string that includes comment characters, like this:

```
                /*
                princ ("/* this is what a comment looks like\n");
                princ ("   when extended to multiple lines\n");
                princ ("*/\n");
    */
```

3.8. Reserved Words

In Gamma, certain words are predefined and reserved for system use. No symbols can be defined by the user that are identical to these reserved words.

The reserved words are:

`class, collect, do, else, for, function, if, local, method, tcollect, while, with.`

For more details see Reserved Words entry in the Reference Manual.

3.9. Memory Management

The programmer generally does not need to consider the memory management aspects of programming in Gamma because Gamma handles all memory management requirements of a task internally through a mechanism known as *garbage collection*. This greatly simplifies programming and eliminates errors associated with dangling pointers, freeing unallocated memory, and array overruns so common in languages such as C. Nevertheless, Gamma provides some functions for examining and invoking the garbage collector. These may be used to determine run-time memory requirements or to ensure that the garbage is collected at pre-determined times.

For more information on garbage collections and the related functions see `gc` in the Reference Manual.

Chapter 4. Tutorial I

This tutorial contains examples of the basic types and mechanisms of Gamma. The first example shows you how to manipulate lists. A list is a very important data type in Gamma. The understanding of list manipulation is a key notion to many of Gamma's constructions. The second example walks you through the famous "Hello world" program.

4.1. Lists

The examples below demonstrate some of the basics of list manipulation. Since Gamma uses the Lisp engine, it inherits the rich set of list functions for which Lisp is known.

```
/* The program starts with the line containing the shell's
directive #!. It makes a Gamma program appear to be a
stand_alone executable to the user. The directive must appear
as the first line and must reference the actual Gamma
executable, not a link.  */

#!/usr/cogent/bin/gamma

/*
 * Lisp defines the functions 'car', 'cdr', and 'cons' as the basic
 * list manipulation functions.  The origins of these names have no
 * meaning on today's computers:
 * CAR - Contents of the Address Register
 * CDR - Contents of the Decrement Register
 * CONS - Construct (OK, this makes sense)
 *
 * We can define alternate names for car and cdr here:
 */

head := car;
tail := cdr;

/*
 * Create some lists.
 */

a = list ("My", "dog", "has", "fleas");
b = list (1, 2, 3, 4, 5, 6);

/* The pound sign "#" in front of a symbol prevents Gamma
from evaluating the symbol so the symbol is taken literally.
Thus, if x = 5, then the function call list (#x, 1) will
create the list (x 1) rather than (5 1).
 */

c = list (#a, #b, #c, #d, #e, #f);
d = list (#d, #h, #b, #i, #g, #e, #c);

/*
 * Take the first component of a list.
 */

e = head (a);
princ ("The first component of ", a, " is ", e, "\n");

/*
 * Take the tail of a list.
 */

e = tail (a);
princ ("\nThe tail of ", a, " is ", e, "\n");

/*
```

```
 * Concatenate two lists.  Notice that list elements do not need to be the
 * same type.
 */

e = append (a, b);
princ ("\nappending ", b, " to ", a, " gives:\n ", e, "\n");

/*
 * Walk a list and print each element
 */

princ ("\nThe elements of 'a' are:\n");
with i in a do
{
    princ (" ", i, "\n");
}

/*
 * Add an element to the beginning of a list
 */

princ ("\nAdding a zero to ", b, "\n");
b = cons (0, b);
princ ("    Gives: ", b, "\n");

/*
 * Take the first element of the tail of the list (the second element)
 * We can use combinations of car and cdr to do this.  However, Gamma
 * predefines a number of car and cdr combinations to make this easier,
 * by inserting multiple 'a's and 'd's between the 'c' and the 'r' in
 * the words car and cdr.
 * For example,   caddr(x)    is the same as    car(cdr(cdr(x)))
 *
 * Gamma defines all one_, two_, and three_letter combinations of
 * car and cdr.
 */

princ ("\na is ", a, "\n");
x = car (cdr (a));
princ ("    car(cdr(a)) is: ", x, "\n");
x = cadr (a);
princ ("    cadr(a) is: ", x, "\n");

/*
 * Some other interesting list functions...
 */

princ ("\n");
princ ("Union of        ", c, " and ", d, " is ", union (c, d), "\n");
princ ("Intersection of ", c, " and ", d, " is ", intersection (c, d), "\n");
princ ("Difference of   ", c, " and ", d, " is ", difference (c, d), "\n");
princ ("Difference of   ", d, " and ", c, " is ", difference (d, c), "\n");
```

4.2. "Hello world" program

This example program prints a personalized "Hello" message a given number of times. It demonstrates variable naming, function definitions and basic control structures, as well as command line arguments.

The program also shows the basic similarities between Gamma and the C language, and some important differences which highlight the power of Gamma.

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

/* This function iterates, saying hello to the given name.
```

```
In Gamma, all functions return a value, which is the
result of the last evaluated expression in the function.
In this case, we return the string "ok".
*/

function say_hello (name, n)
{
    local i;

    for (i = 0; i < n; i++)
        princ ("Hello ", name, "\n");
        "ok";
}

/* A program may optionally have a function main()
declared, in which case Gamma will execute the
main() function after all of the program file has been
read in. If no main() declared, the programmer must
explicitly call a function or enter an event loop at
some point while reading the file. Any code which is not a
function definition will be automatically run AS THE FILE
IS READ. This is useful for providing feedback to the user
while loading.  */

function main ()
{
    /* Define some locally scoped variables. Notice that
    Gamma implements abstract data typing, so it is not
    necessary to declare the variable type.
    */

    local repeat = 1, my_name = "world";

    /* Access the command line arguments. argv is a list of the
    command line arguments, as strings, where the first argument
    is the program name.  */

    /* The second argument (cadr(argv)) is my_name,
    if present.
    */

    if (cadr(argv))
        my_name = cadr (argv);

    /* The third argument (caddr(argv)) is the number
    of iterations, if present.
    */

    if (caddr(argv))
        repeat = number (caddr(argv));

    /_now print the message */

    result = say_hello (my_name, repeat);
    princ (result, "\n");
}
```

Chapter 5. Control Flow

Gamma provides a variety of mechanisms for control flow. These generally fall into the categories of statements, function calls, event handlers and error handlers. Many of these are dealt with in more detail in other sections of this document. All functions have a detailed entry in the Reference Manual.

5.1. Statements

A *statement* in Gamma is any syntactically complete piece of code that can be independently evaluated, written in statement syntax. There are two kinds of statement syntax, as follows:

; A semi-colon at the end of an expression is used to denote a single, one-line statement.

{ } Curly brackets surrounding zero or more expressions or statements create a single statement from them. Multiple statements within the curly brackets can be grouped in any combination, and nested in any number of levels. This statement syntax is also referred to as a *compound statement* or *code block*. Each expression or statement is evaluated in order, and the result is the result of the last statement or expression. A code block in Gamma does not create a local scope.

Gamma has a number of built-in statements, several of which are explained below. For a complete list of built-in Gamma statements, see Statements in the Reference Manual.

5.1.1. Conditionals

Gamma contains a single conditional statement: `if`. The `if` function evaluates its condition, and if the condition is non-`nil`, then the first statement after the `if` is performed. If the condition is `nil`, the `else` clause of the statement is executed. It is not mandatory that an `if` statement have an `else` clause. In the case of nested `if` statements, an `else` clause will always bind with the nearest `if` statement.

For example:

```
...
if (var == 1)
    count1 ++;
else if (var == 2)
    count2 ++;
else if (var == 3)
    count3 ++;
...
```

The `if` statement accepts only a single statement for its true and false clauses. Using the code block statement syntax, it is possible to perform more than one action inside an `if`.

Example:

```
...
if (var == "string")
{
     count++;
     princ("The number is ", count, "\n");
}
...
```

5.1.2. Loops

Gamma supports three looping statements: `for`, `while` and `with`. The `for` loop looks exactly like a `for` loop in C:

```
...
local i;
for (i = 1; i < N; i++)
 {
   body_statements
 }
...
```

A `while` loop is also exactly like the C `while` loop, though the do/while variant available in C is not supported in Gamma.

Gamma adds the `with` loop, which walks a list or an array and executes a body statement once for each element in the list or array. The `with` loop may be instructed to collect the results of the body statement for each iteration, and return the accumulated results as a new list or array. The iteration variable in a `with` loop is defined only within the body of the loop. For an example, see the Lists and Arrays section in the Basic Data Types and Mechanisms chapter.

5.1.3. Goto, Break, Continue, Return

Gamma does not contain facilities for non-linear local jumps. Many languages provide one or more wrappers on the `goto` function, such as `break` (go to the end of the expression), `continue` (go to the beginning of the expression) and `return` (go to the end of the function). These facilities can be used on occasion for clarity, but can commonly act to confuse both the programmer and the reader.

In Gamma, such a facility would be very confusing, as it is not always clear which expression constitutes the scope of a break, continue or return. In addition, the execution speed penalty associated with supporting local jumps in a functional language generally outweighs the benefit in the few cases where a local jump would be convenient. Further, because Gamma is dynamically scoped, a local jump would be defined as a jump that does not cross a scope boundary rather than the more common C definition of a jump that does not cross a function boundary. This distinction greatly reduces the value of, and the need for, a local jump capability.

The most common local jump instruction in procedural languages is `return`. This instruction moves the execution to the bottom of the function and supplies a return value for the function. In Gamma, all functions implicitly return a value which is the result of the last expression to be evaluated within the function body. If no expression is evaluated, then the function returns `nil`. If the programmer wishes to return the value of a symbol, he or she can simply write that symbol, followed by a semicolon, as the last statement to be evaluated in the function.

5.2. Function Calls

Any Gamma expression which makes a call to a function, such as `tan(3.14159)`, causes a change of program flow, entering a new scope and causing execution to be temporarily diverted into the function being called. In most cases, the function simply returns and flow continues at the expression containing the function call.

If an error occurs during a function call, the function will not return, and execution will continue from the most recent `protect`/`unwind` or `try`/`catch` construct (see the Error Handling section of this chapter).

5.3. Event Handling

An *event* is a change to which the system responds. Events include interprocess communication (message exchanging between processes) events, signals, timer events (execution of a block of code at specified time), Graphical User Interface events (clicking on the screen buttons), and DataHub exceptions (a change in value of a DataHub point).

Gamma provides a generalized event handling capability which processes all these events. A Gamma function that responds to an event is called an *event handler*, or a *callback*. A pair of functions, `next_event` and `next_event_nb`, invoke the event handler, that is, any callback function(s) that have been attached to the event. For example, the simple construct:

```
while ( t)
{
    next_event ( );
}
```

placed at the end of the file, together with the set of callback functions defined in the application to handle all the required events, creates a purely event-driven Gamma program. However, unlike typical event-driven systems applications (such as many GUI-based applications), the user has here the opportunity to further process the events, providing a conditional which is more complex than an infinite loop, or even choosing not to receive events. The events will not be processed (or callbacks invoked) until one of the `next_event` calls are made.

The result of `next_event` is the result of executing the callbacks, or `nil` if no event handler has been defined (that is, no callbacks have been attached). The result of `next_event_nb` (nb stands for non-blocking) is the same except that `nil` is also returned if no event was available.

Attaching callbacks and receiving events depends on the type of event. For example, in Photon the function `PtAttachCallback` is used to attach actions to clicking on buttons. Normally one does not wait for an explicit event type, that is, the `next_event` call will process ANY event defined within the system. The following sections describe how some common event types are handled.

5.3.1. Interprocess Communication Message Events

Gamma uses a *send/receive/reply* (SRR) mechanism to provide interprocess communication via *messages*. In this context, a message is any valid Gamma or Lisp expression passed synchonously from one process to another, using the `send` function. The engine treats the message as a null-terminated character string in Lisp syntax (Gamma's internal representation), which it parses and evaluates. Any expression may be transmitted in this way, including function definitions, function calls, variable names and complex blocks of code. The reply returned is the result of the evaluation.

This process is transparent to the application. The Gamma engine evaluates the incoming message inside an implicit `try`/`catch` block to ensure that externally originated expressions cannot accidentally affect the running program. Any errors that occur while the message is being evaluated will be indicated in the return value for the message, but the overall running status of the engine will not be otherwise affected.

For more information see the Interprocess Communication section in the Special Topics chapter.

5.3.2. Timers

A *timer* is a Gamma expression that is submitted for evaluation at specified time in the future. Timer related events are set up through the `every` and `after` functions which provide a relative time delay, and the `at` function which accepts an absolute time. These functions accept timing parameters and a block of code that will be evaluated when the timer expires. The code can be simply a function name, or can be an entire expression to be evaluated.

A timer will only be handled during a call to `next_event`, `next_event_nb` or `flush_events`. If a timer expires while another operation is being performed, the engine will evaluate the timer code at the next event handling instruction.

In Gamma, by default, timers are internally handled by using *proxies*. A proxy is non-blocking system message that does not require a reply. Gamma can act on a proxy immediately, or delay a little while in order to finish what it is currently doing. It is this 'little while' that becomes the limit of the accuracy of the timers in Gamma when they are driven from proxies. You see, most programs written in Gamma are run through an event loop. The maximum time a proxy-based timer can be delayed is the execution time of the longest path of code attached to an event. Since this quantity is totally dependent on how your code is written and the speed of your CPU, it is difficult to put an exact number on the practical accuracy of Gamma's proxy-based timers. The maximum resolution of the timers is equal to the OS tick size setting.

5.3.2.1. Setting a timer

A timer is created when any of the following functions are called:

The `after` timer is used to evaluate a piece of code after a given amount of time. It is a one-shot timer that " goes away " after being fired.

The `every` timer is used to evaluate a piece of code at a specified interval.

The `at` timer evaluates the associated code when the current time matches the profile created by six arguments to the function.

For syntax and examples see the Reference Manual.

5.3.2.2. Canceling a Timer

All timer functions return a timer_id that can be used to cancel the timer. To do so, the `cancel` function is called, using the timer-id for its argument.

5.3.2.3. The TIMERS variable:

On startup of Gamma a variable called `TIMERS` is initialized. This variable is an array of all the timers currently in the system. Initially, its value is [ ] (an empty array) but as timers are added to the system this array grows.

```
Gamma> TIMERS;
[]
Gamma> a = every(3,#princ("Hello\n"));
1
Gamma> >TIMERS;
[[860700531 176809668 3 ((princ "Hello\n")) 1]]
```

The `TIMERS` variable was initially an empty array. Once the first timer was added the `TIMERS` variable contained information on the first timer. The contents of each element can be summarized as:

```
second nanosecond repeat function timer_id
```

The second is the number of seconds since Jan. 1, 1970 which is compatible with the `clock` and `date` functions. The nanosecond is the fraction of the second. Combining these two times gives an accurate time that identifies the next time the timer will fire. The repeat is `nil` if the timer is a once-only "`after`" timer or an "`at`" timer. Otherwise, this is the period of an "`every`" timer. The next item is the code that is evaluated when the timer fires. The last item in this sub-array is the timer-id.

This array grows as timers are added to the system:

```
    Gamma> b = every(5, #princ("Test\n"));
    2
    Gamma> _timers_;
    [[860700531 176809668 3 ((princ "Hello\n")) 1]
[860700563 494732737 5 ((princ "Test\n")) 2]]
    Gamma>
```

An important point to remember is that the `TIMERS` variable is an array, and therefore can be referenced as such. To reference the first element in the `TIMERS` array, use `TIMERS` [0] just like a regular array reference. Altering the `TIMERS` array can cause unpredictable behavior, and should be avoided.

5.3.2.4. Blocking timers from firing

Sometimes a segment of code is written which must be executed non-stop, without an interruption from timers. To accomplish this, the code is wrapped between `block_timers` and `unblock_timers` functions. In this way the code will be safe from interruption from timers. This blocking is not necessary unless timers have been bound to signals rather than proxies.

5.3.2.5. `timer_is_proxy` function

This function controls how timers are fundamentally handled within Gamma. By default, timers are handled by the processing of proxies. This allows Gamma to delay the timer, if necessary, when a critical system process is occurring.

Calling the `timer_is_proxy` function with `nil` makes all timers operate by using signals. In the QNX 4 operating system, for example, SIGALRM is used, and the attached code is run as a handled signal. Running timers via signals has some very dramatic consequences. First and foremost, when running in this mode *all timer code must be signal safe*. This status must be analyzed with caution, as most code is *not* signal safe. This mode should be avoided except in very rare circumstances.

5.3.3. Symbol Value Events (Active Values)

Gamma has the capability to generate an event when a variable is modified in any way. A variable that can trigger an event is called an *active value*. The code that is executed when the variable changes is called the *set expression* that is effectively a callback.

The `add_set_function` permits attaching an expression to any defined variable. (It is an error to attach a set expression to a constant symbol.) When the value of a symbol changes, either by being declared within a sub-scope or by being explicitly changed using `=`, `:=`, `--`, or `++`, the Gamma engine checks the symbol for the existence of an associated set expression. If a set expression exists, then it is executed directly after the value of the symbol is changed. This set expression can be any valid Gamma code, and is evaluated within its own sub-scope. The symbols `value`, `previous` and `this` are defined within this sub-scope to be the current value of the symbol, the previous value of the symbol, and the symbol itself respectively. If an active value causes another active value to change, then that new active value's set expression is also evaluated. This provides a very simple and powerful means by which forward chaining algorithms such as those in expert systems can be implemented.

Active values provide a very powerful way of constructing an event-driven application with a high degree of cohesiveness. Often, there is some functionality that is related to a derived variable, not an event itself. The function can be attached to that internal variable, decoupling it from the event, and generating clearer and more concise code. In the following example, we wish to attach an action to an alarm condition, which in turn is generated by a change in a measured variable (e.g., the temperature). The `#` sign protects an expression from evaluation, causing it to be treated as a literal. (We discuss Deferring Expression Evaluation in more details later, in the chapter on Advanced Types and Mechanisms).

```
  Gamma> temp = 35;
  35
  Gamma> hi_limit = 40;
  40
  Gamma> function check_temp (tp)
   {if (tp > hi_limit) alarm_hi_on = t;
    else alarm_hi_on = nil;}
  (defun check_temp (tp)
(if (> tp hi_limit)
  (setq alarm_hi_on t)
  (setq alarm_hi_on nil)))
  Gamma> function print_alarm ()
   {if (alarm_hi_on == t) princ("Alarm is on: Temp is over ",hi_limit, ".\n");
    if (alarm_hi_on == nil) princ("Alarm is off.\n");}
  (defun print_alarm ()
(progn (if (equal alarm_hi_on t)
          (princ "Alarm is on: Temp is over " hi_limit ".\n"))
       (if (equal alarm_hi_on nil)
          (princ "Alarm is off.\n"))))
  Gamma> add_set_function(#temp,#check_temp(temp));
  (check_temp temp)
  Gamma> add_set_function(#alarm_hi_on, #print_alarm());
  (print_alarm)
  Gamma> temp = 38;
  Alarm is off.
  38
  Gamma> temp = 39;
  39
  Gamma> temp = 42;
  Alarm is on: Temp is over 40.
  42
  Gamma>
```

5.3.4. Cogent DataHub Point Events (Exception Handlers)

The Cogent DataHub is a data collection and distribution center designed for easy integration with Gamma. A point is a name for data in the Cogent DataHub. An exception handler is a Gamma expression that is attached to a point. The Cogent DataHub can asynchronously transmit any number of point values to a Gamma program, which will then automatically update the value of a symbol named the same as the DataHub point.

The `add_exception_function` and `add_echo_function` functions permit an application to bind an expression to a DataHub point exception in a similar way to `add_set_function` above. If an application can both write a point and receive exceptions from that point, then the DataHub will "echo" the exception originated by the application. The two functions make it possible to distinguish between these two situations. Only the originating task will see a point exception as an echo, while all other tasks will see a normal exception.

In addition, the programmer may attach any number of expressions to the symbol, to be evaluated when the DataHub point changes. The expressions are evaluated within a sub-scope, with the special symbols: `value`, `previous` and `this` defined to be: the current value of the point, the previous value of the point and the point name itself as a symbol. An exception handler is only triggered during a call to `next_event`, `next_event_nb` or `flush_events`.

5.3.5. Windowing System Events

Gamma's GUI support offers `PtAttachCallback` for Photon and `AttachCallback` for X Windows that permit attaching callbacks to any GUI event. Like the other event handling functions, the user can in fact bind any Gamma expression for execution when the event occurs.

5.3.5.1. GUI Event Handlers (Callbacks)

A GUI event handler, also known as a *callback*, is an arbitrary expression attached to a particular callback of a widget. A callback may occur whenever a call is made to `next_event`, `next_event_nb`, and `flush_events`. If the appropriate GUI event has occurred, then the Gamma engine automatically evaluates any callback handlers that deal with the event attached to the particular widget. This results in essentially asynchronous program flow, where the callback may occur, from the user's perspective, at any time during the program execution. In reality, the GUI event is only handled if the system has been instructed to deal with one or more incoming events.

5.3.6. Signals

Signals are the traditional method of asynchronous communication between tasks, in which no data is transferred. A *signal handler* is an expression attached to an operating system signal, which is delivered asynchronously to the running program. A signal handler is attached by a call to the `signal` function.

A signal pre-empts any activity except garbage collection, and causes control flow to enter the signal handler. The signal handler should not call non-reentrant functions. It is safe for a signal handler to make a call to the `error` function, which will throw flow control to the nearest error handler.

Gamma supports the following signals:

```
SIGABRT, SIGBUS, SIGCHLD, SIGCONT, SIGDEV,
SIGEMT, SIGFPE, SIGHUP, SIGILL, SIGINT,
SIGIO, SIGIOT, SIGKILL, SIGNAL_HANDLERS,
SIGPIPE, SIGPOLL, SIGPWR, SIGQUIT, SIGSEGV,
SIGSTOP, SIGSYS, SIGTERM, SIGTRAP, SIGTSTP,
SIGTTIN, SIGTTOU, SIGURG, SIGUSR1, SIGUSR2,
SIGWINCH
```

For the description of signal values see the `signal` entry in the Reference Manual.

5.3.6.1. `block_signal` & `unblock_signal`

There are times when certain portions of code must not be interrupted by certain or all signals. Use the `block_signal` and `unblock_signal` functions to protect a process.

5.4. Error Handling

An error handler is a function that responds to an error. In general, when a program executes, the flow of control moves from one expression to the next, possibly passing into and out of function calls, and following branches and loops as necessary. If an error occurs, however, the flow of control will not move to the next expression, but will jump immediately to the most recently declared error handler, either through the `protect`/`unwind` or the `try`/`catch` constructs. These constructs are the two pairs of functions available in Gamma which allow for trapping and handling errors. See Tutorial II for more details.

The combination of signal handlers and error handlers can cause a program to jump to a predefined point at any time during its execution. An error can be explicitly caused by a call to the `error` function.

5.4.1. Situations that might cause Gamma to crash

Gamma is a very robust language, particularly in comparison to programming in C. However, the power and ease of use can sometimes lead a programmer to create situations that could crash the Gamma engine. Generally these are errors that would certainly have crashed a C program, and would be

considered part of the debug cycle. The following list highlights some situations where care must be taken:

1. A call to `init_ipc` inside a timer or signal handling routine. You should call (`init_ipc`) once at the beginning of your program if at all.
2. Abuse of Photon widget resources. While care has been taken to minimize the risk of a crash by abusing the Photon widgets, the bottom line is that widgets are raw C structures with fairly complex manipulation functions. If you write a bad value into a C structure, your program will crash. If you fail to call `PtInit`, your program will crash. If you create a non-window widget with no parent, your program will crash. These are just facts of life.
3. Gamma provides a 'wrapper' for most of the standard C library functions that makes the corresponding C function call after extracting the Gamma arguments. If the arguments passed cause the C function to crash, then your program and the Gamma engine will crash as well.

Having said these things, we are always interested in hearing about new ways that we can make Gamma more robust. Please don't hesitate to let us know if you find a weak spot.

Chapter 6. Tutorial II

This tutorial includes examples related to control flow in Gamma: namely, error handling and dynamic scoping.

6.1. Error Handling - `try`/`catch`, `protect`/`unwind`

This example demonstrates the error handling mechanisms available in Gamma. There are two basic means of trapping and handling errors.

1. Execute a protected block of code, and specify an error handler which is only executed if an error occurs within the protected code. If an error occurs, the error handling code is executed, and the error condition is cleared. This is the `try`/`catch` mechanism.
2. Execute a protected block of code, and specify a second block of code which must be executed even if an error occurs in the first block. Normally when an error occurs, the execution stack is unwound to the nearest error handler, aborting any intervening execution immediately. If a block of code must be run, even when an error occurs, we want to unwind protect that code. After the error is dealt with, it is passed on up the stack rather than being cleared. This is the `protect`/`unwind` mechanism.

The code for the example is shown below.

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

/*
 * Create a function which has an error in it.
 * The symbol zero is not defined.
 */

function sign (x)
{
    if (x < zero)
        princ ("Negative\n");
    else
        princ ("Positive\n");
}

/*
 * Create a function which checks the sign of a number,
 * but ensures that an error will not terminate the program.
 */

function checkit (x)
{
    princ ("\nEntering a TRY, CATCH block...\n");
    try
    {
        sign (x);
    }
    catch
    {
        princ ("Oops: ", _last_error_, "\n");
    }
}

/*
 * Create a function which checks the sign of a number,
 * and which will print a status message whether or not an error
 * occurs, before passing a possible error condition up the stack.
 * The 'princ' statement in this case will always be run, even if
 * an error occurs.
 */
```

```
function unwindit (x)
{
    princ ("\nEntering a PROTECT, UNWIND block...\n");
    protect
    {
        sign (x);
    }
    unwind
    {
        princ ("Finished checking the sign\n");
    }
}

/*
 * Attempt to call this function, but if we get an error,
 * simply print the error message and continue.
 */

checkit (-5);

/*
 * Run the same code, but with zero defined
 */

zero = 0;
checkit (-5);

/*
 * Run the unwind protected function.  This should print its unwind
 * message.
 */

unwindit (-5);

/*
 * Make 'zero' undefined again so that the error will occur.  Now we
 * run a function which is unwind protected.  This function will not
 * return since it passes the error up to the global error handler,
 * which causes the program to exit.
 */

zero = _undefined_;
unwindit (-5);

princ ("This message should never be printed\n");
```

6.2. Dynamic Scoping

This example uses the error handling mechanisms from the previous Error Handling section to demonstrate dynamic scoping. Most compiled languages use lexical scoping, which means that a variable is defined only where it is visibly declared, either as an external global, file global, or local variable. Gamma uses dynamic scoping, meaning that a variable is defined in any function which defines it, and in any function which the defining function subsequently calls. This powerful mechanism allows the programmer to override global variables by defining them in a higher scope, and then calling a function which believes itself to be using a global variable.

One useful side-effect of dynamic scoping is that functions and variables do not have to be declared before they are used in other functions. The function or variable only has to be declared when the other function is actually run.

The code for the example is shown below.

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

/*
 * Create a function which has an error in it.
 * The symbol zero is not defined.
 */

function sign (x)
{
    if (x < zero)
        princ ("Negative\n");
    else
        princ ("Positive\n");
}

/*
 * Create a function which checks the sign of a number,
 * but ensures that an error will not terminate the program.
 */

function checkit (x)
{
    try
    {
        sign (x);
    }
    catch
    {
        princ ("Oops: ", _last_error_, "\n");
    }
}

/*
 * Create a function which locally declares the variable 'zero', and
 * then calls the checkit function.  Since 'zero' is a local variable,
 * the local value will override the current global definition, which
 * is undefined.
 */

function zero_check (x)
{
    local zero = 0;
    checkit (x);
}

/*
 * Create a function which sets zero to -10, and calls the checkit
 * function.
 */

function minus_ten_check (x)
{
    local zero = -10;
    checkit (x);
}

/*
 * Attempt to call the checkit function with zero not defined.
 */

princ ("With 'zero' undefined...\n");
checkit (-5);

/*
 * Now let zero be defined and try again.
 */

princ ("\nWith 'zero' locally defined to 0...\n");
```

```
zero_check (-5);

/*
 * Now run with zero defined as -10
 */

princ ("\nWith 'zero' locally defined to -10...\n");
minus_ten_check (-5);

/*
 * Finally, try running checkit again from the global scope.  Note that
 * zero is undefined once again.
 */

princ ("\nOnce again from the global scope...\n");
checkit (-5);
```

6.3. Error Handling - interactive session

The code below provides an example of starting an interactive session in case of an error. In this example a window with two buttons is created. Pressing the button labeled good button will print a message to stdout. Pressing the button labeled error button will cause the UNDEFINED symbol g to be evaluated. This causes an error and starts the interactive session from the catch block. The function that runs the interactive session is designed to be recursive and relies on the dynamic scoping of variables in Gamma. Notice that Lisp grammar is being used in this interactive session. You can query the value of any variable simply by typing the variable name in. For example:

```
win
but1
but2
(@ win title)
```

or use some functions like:

```
(stack)
(* 8 3)
```

Before terminating the interactive session, try to resize the window. Notice that it does not work because the event loop is temporarily suspended. Now exit the interactive session by typing **Ctrl - D** and notice that the window can now be resized. Also notice that once the event loop is re-started, the contents of the window are not updated but the callbacks are still active. This happens because Photon was interrupted in the middle of a function call and an error condition now exists between Gamma and the Photon library.

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

require_lisp("PhotonWidgets");

PtInit(nil);
win = new(PtWindow);
win.SetArea(100,100,100,100);

but1 = new(PtButton);
but1.text_string = "good button";
PtAttachCallback(but1,Pt_CB_ACTIVATE,#princ("good button\n"));
but1.SetPos(10,10);

but2 = new(PtButton);
but2.text_string = "error button";
PtAttachCallback(but2,Pt_CB_ACTIVATE,#g);
but2.SetPos(10,40);
```

```
PtRealizeWidget(win);

function interactive_mode ( level )
{
 local expr;

    princ("internal error: ", _last_error_,"\n");
    writec(stdout,"\ndebug", level,">");
    while ( (expr = read(stdin)) != _eof_)
 {
        try
        {
            writec(stdout,eval(expr));
            writec(stdout,"\ndebug", level,">");
        }
        catch
        {
            interactive_mode(level + 1);
            writec(stdout,"\ndebug", level,">");
        }
    }
}

while (t)
{
    try
    {
        next_event();
    }
    catch
    {
        princ("starting temporary interactive mode using Lisp grammar\n");
        princ("use ^D to exit this mode and return to event loop\n");
        interactive_mode(1);
        princ("\nleaving temporary interactive mode\n");
    }
}
```

Chapter 7. Functions and Program Structure

7.1. Function Definition

A function is defined in Gamma using a `function` statement:

```
Gamma>  function thing (a) { random() * a;}
(defun thing (a) (* (random) a))
Gamma>  thing (5);
2.3869852581992745399
```

```
function pow (v, exp)
{
    local result = 1;

    while (exp -- > 0)
    {
    result *= v;
    }

    result;
}
```

No type specification is used since the type returned will be determined by the expressions within the function when it executes:

```
Gamma> pow (2,3);
8
Gamma> pow (2.1, 3.25);
19.4481
```

C programmers should note that this typeless function definition bears no similarity to a void function type, which does not exist in Gamma.

The value and type of a function is the return value of the last expression evaluated within the function. To return the value of a specific variable that only exists within the scope of the user function, that variable name is placed by itself on the last line of the function. This effectively causes that symbol to be evaluated, returning its value.

Functions do not need to be defined before they are referenced in a file, but they must exist before they are called. In other words, it is the run-time order, not the loading (reading) order that is important.

7.2. Function Arguments

When a function is called, the arguments in the call are mapped to the arguments specified in the function definition on a one-to-one basis. The Gamma engine evaluates the arguments and maps the results of those evaluations to each function argument name. Since Gamma is abstractly typed, there is no need to specify a data type in the function definition. If a particular data type is required within the function, then the function body can check for the type using the type predicate, *`type`*`-p`.

7.2.1. Variable number of arguments

It is possible to create a function that takes a variable number of arguments. The last argument in a function’s argument list may be made to act as a "catch-all" or *vararg* argument which collects all remaining arguments provided in the function call as a list. For example,

```
function f (x, y...)
```

creates a function with 2 mandatory arguments, the second of which can have one or more values. If this function is called as `f (1,2)`, then *x* will have the value `1`, and *y* will have the value `(2)`, that is, a list containing one element whose value is `2`. If this function is called as `f (1,2,3,4,5)`, then *x* will be `1`, and *y* will be `(2 3 4 5)`, a list of four elements. If this function is called as `f(1)`, then an error would occur because y is not optional.

7.2.2. Optional arguments

Gamma allows optional arguments at the end of an argument list. An optional argument is specified by appending a question mark (`?`) to an argument in the function's argument list. All arguments after the first optional argument are implicitly optional as well. If the caller wants to provide a value to an optional argument, then the caller must also provide values for all preceding optional arguments. If an optional argument is not provided during the call, then the argument will take on the value `UNDEFINED`, which must be dealt with within the body of the function. A default value for an optional argument can also be provided in the function definition. For example,

```
function f (x, y?, z=5)
```

creates a function with 1 mandatory argument and two optional arguments. The argument *y* has no default value, and *z* has a default value of `5`. This function could be called as `f(1)`, `f(1,2)` or `f(1,2,3)`.

7.2.3. Protection from evaluation

Any function argument can be protected from evaluation by an exclamation mark (`!`) before the argument name in the function's argument list. For example,

```
function f (x, !y)
```

creates a function with two mandatory arguments , the second of which will not be evaluated when it is called. If this function were called as `f (2+2, 3+3)` then *x* would have the value of `4`, and *y* would have as its value the expression `3+3`. *y* could be evaluated using `eval(y)` to produce the value `6`.

7.2.4. Variable, optional, unevaluated arguments

A variable argument can also be made optional. If so, and if it is not evaluated, then all the arguments which are collected into its list will not be evaluated either. For example,

```
function f (!y...? = 17)
```

creates a function with one optional argument named *y*. The argument *y* is not evaluated, and may take any number of values, passed as a list. If no argument is specified to the function, then *y* will have the value of `17`. If, instead of `17` the default is set to `nil`, no default will be assigned. This syntax effectively gives a way to pass a list of arguments of any length to a funtion.

7.2.5. Examples

The following program shows example functions with argument lists similar to those described above.

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

function variable_args (x, y...)
{
  princ("---- Output from variable_args(x, y...) ---- \n");
  princ("The first arg: ", x, "\n");
  with a in y do
    {
      princ("One of the variable args: ", a, "\n");
    }
  princ("\n");
}
```

```
function optional_args (x, y?, z=5)
{
  princ("---- Output from optional_args(x, y?, z=5) ---- \n");
  if (undefined_p(y))
    y = "This value has not been defined.";
  princ("The first arg: ", x, "\n");
  princ("The second arg: ", y, "\n");
  princ("The third arg: ", z, "\n");
  princ("\n");
}

function no_eval(x, !y)
{
  princ("---- Output from no_eval(x, !y) ---- \n");
  princ("This argument was evaluated: ", x, "\n");
  princ("This argument was not evaluated: ", y, "\n");
  princ("\n");
}

function many_args (fixed_arg, !y?... = nil)
{
  princ("---- Output from many_args(fixed_arg, !y?... = nil) ---- \n");
  princ("fixed_arg: ", fixed_arg, "\n");
  princ("y: ", y, "\n");
  princ("The first y arg: ", car(y), "\n");
  with a in cdr(y) do
    {
      princ("The next y arg: ", a, "\n");
    }
  princ("\n");
}

variable_args("hello", 9, "world", 4 + 7, #x);
optional_args(1);
optional_args(1, 2);
optional_args(1, 2, 3);
no_eval(2+2, 3+3);
many_args("Fixed", "hello", 9, "world", 4 + 7, #x);
```

The output of this program is as follows:

```
    ---- Output from variable_args(x, y...) ----
    The first arg: hello
    One of the variable args: 9
    One of the variable args: world
    One of the variable args: 11
    One of the variable args: x

    ---- Output from optional_args(x, y?, z=5) ----
    The first arg: 1
    The second arg: This value has not been defined.
    The third arg: 5

    ---- Output from optional_args(x, y?, z=5) ----
    The first arg: 1
    The second arg: 2
    The third arg: 5

    ---- Output from optional_args(x, y?, z=5) ----
    The first arg: 1
    The second arg: 2
    The third arg: 3

    ---- Output from no_eval(x, !y) ----
    This argument was evaluated: 4
    This argument was not evaluated: (+ 3 3)

    ---- Output from many_args(!y?... = nil) ----
```

```
One of the args: hello
One of the args: 9
One of the args: world
One of the args: (+ 4 7)
One of the args: 'x
```

7.3. Function Renaming

When a function is defined, Gamma automatically assigns the function definition to the symbol that was provided as the function name, in the global scope. This does not mean that the symbol and the function definition are permanently related.

(A function definition is an independent data object which can be passed as an argument to a function or assigned to a symbol in any scope. For a C programmer this makes a Gamma function definition operate in much the same way as a function pointer in C. However, Gamma function definitions are much more versatile.)

It is possible to re-map a function definition at run-time to modify its behavior. For example, we may like to modify the function `pow` defined in the Function Definition section. We would like the improved `pow` to check the argument type and accept a string as its argument as well as a number. In Gamma, there are functions like `int_p`, `real_p`, and `string_p` that are used to determine the data type of a variable. (For the complete list of `-p` functions see Data Types and Predicates in the Reference Manual).

Thus, we rename the `pow` function defined in the section Function Definition to `__pow` and write the new version as follows:

```
...
__pow = pow;
function pow (v, exp)
{
    local result;
    if (string_p (v))
    {
        result = "";
        while (exp -- > 0)
            result = string(result, v);
    }
    else
        result = __pow(v, exp);
    result;
}
```

Then the function call `pow("hello", 3)` will produce:

```
"hellohellohello"
```

7.4. Loading files

Files are loaded from the disk to memory using the `load` and `require` functions. The `load` function loads a Gamma file every time it is called. The `require` function checks to see if a Gamma file has been loaded, and if not, it loads it. The `load_lisp` and `require_lisp` functions do the same thing for files written in Lisp grammar. All of these functions take the name of the file, as a string, for their argument.

As a file is loaded by the Gamma engine, the `require` mechanism is used to access additional files. This is similar to the `#include` directive used in C programs, and likewise permits modularization of the application code. Note however that since Gamma is a run-time language, there is no equivalent to

object modules of compiled languages. The `require` function therefore provides the sole mechanism for bringing together modules that define an application.

The pre-defined global variable `_require_path_` contains a list of the paths to be searched to find the specified filename. This variable usually references the current directory, and the location for libraries. The list of paths can be augmented with:

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

The pre-defined global variable `_load_extensions_` contains a list of default extensions that are used by the `require` functions. Filenames with these extensions do not have to specify the full filename in the `require` argument. The variable is initialized to (`".slg" ".lsp" ""`), and can be augmented in the same way as `_require_path_`.

7.5. The `main` Function

In Gamma there is no requirement for a function named `main` as there is in C. As a program file is loaded, a call to a function at the outermost scope will in fact cause that function to be run at that point. In the same way, variable definitions and assignments at the outermost scope level are executed, effectively becoming globals. In most cases, the application is initiated by calling the user's "top level mainline" function at the end of the file, or by entering a loop, such as an infinite event loop.

If a function is defined with the name `main`, then Gamma will automatically start to execute that function after the load is complete. This is equivalent to placing `main` as the last statement in the file. Note that `main`'s function definition must contain the keyword `function` , just like any other function definition. Since `argv` is available as a global variable, `main` does not require any arguments.

7.6. Executable Programs

Since the Gamma language is based on Lisp (ie. SCADALisp), programs can be written and executed using either Gamma or Lisp grammar. How to execute a Gamma program is discussed in Stand_Alone Executable Programs in the Getting Started chapter of this Guide, as well as in the next two sections of this chapter.

Writing Lisp programs is beyond the scope of normal Gamma programming, but it may be useful from time to time to invoke a Lisp executable. This is done in a similar way to Gamma. Stand-alone programs will invoke the Lisp engine by using the following shell directive as the first line of the file:

```
#!/usr/cogent/bin/lisp
```

The Photon dialect is available through:

```
#!/usr/cogent/bin/phlisp
```

7.7. Running a Gamma Program

Gamma programs can be run in two ways: as a stand-alone executable, or by invoking Gamma from the command line.

1. For stand-alone executable program, the user simply types the name of the executable, possibly with command line arguments. The program invokes the Gamma engine through the shell `!#` directive (as explained in Executable Programs in the Getting Started chapter of this Guide.
2. In the case where there is no Gamma engine "embedded" into the program, the command **gamma** is available to run the executable. This command has several options, a few of which we mention here, and the rest of which are given in the **gamma** entry in the Reference.

`-h` gives a help message displaying all the options for this command.

`-C` declares all Gamma constants at startup. These constants can be viewed using the `apropos` function.

`-d` saves debugging information: the file name and line number.

`-F` declares all Gamma functions at startup. As with the `-C` option, these functions can be viewed using the `apropos` function.

The next section discusses command line arguments for a program, and how to access them within the program.

7.8. Command Line Arguments

Gamma provides a mechanism for accessing command line arguments. The symbol `argv` contains a list of the parsed command line arguments. Thus, if you have an application named **my_app** which takes two arguments `arg1` and `arg2`, then the executable invoked with:

```
my_app arg1 arg2
```

will receive the following `argv`:

```
(my_app arg1 arg2)
```

Like any list, the length of `argv` is simply `length (argv);` The command line arguments can be accessed like any list, using any of the following sample approaches:

```
for ( i=0; i < length(argv); i++)
{
    arg = car (nth_cdr (argv, i));
    ... process arg ...
}
```

or similarly, but more efficient:

```
while (length (argv) > 0)
{
    arg = car (argv);
    ... process arg ...
    argv = cdr (argv);
}
```

which can also be expressed, still more efficiently, and without modifying the original `argv`, with:

```
for (i=argv; i; i = cdr(i))
{
    arg = car (i);
    ... process arg ...
}
```

Chapter 8. Object Oriented Programming

Classes are a powerful feature that helps users to organize a program as a collection of objects. Users that are familiar with C++ will find some syntax similar.

8.1. Classes and Instances

A class is a collection of variables and functions that, together, embody the definition of data type that is distinct and significant for the user's problem. Class functions are called *methods*. Class variables can be of the two kinds: *attributes* and *class variables*. Attributes are more common and do not require any special identifiers. Class variables are defined with the identifier static. We'll discuss class variables later in this chapter. Every class has a name. Here we define a class named `Polygon`, and give it four attributes:

```
class Polygon
{
    sides;
    angles;
    dimensions = 2;
    color = nil;
}
```

Here is another example, `Catalog`, with one attribute, defined in interactive mode:

```
Gamma> class Catalog { data;}
(defclass Catalog nil [][data])
```

When you define a class in interactive mode Gamma returns an internal representation of its *class definition* in Lisp syntax. This definition is a list with the following elements: the `defclass` function, the class name, the parent class name (`nil` in this case), the class methods and class variables in one array (none in this example), and the class attributes in a second array.

Default values can be assigned to the attributes. For example, the `Polygon` class (above) has 2 dimensions and no color by default. The `Catalog` class has no default data values.

8.1.1. Instances

A class is an abstract data type. A class is used by constructing new *instances* of it. This is done using the function `new`:

```
Gamma> pentagon = new(Polygon);
{Polygon (angles) (color) (dimensions . 2) (sides)}

Gamma> autoparts = new(Catalog);
{Catalog (data)}
```

In this example, the class is `Polygon` and the newly-created instance of the class is `pentagon`. Or, the class is `Catalog` and the instance is `autoparts`.

The variables of the instances of a class are called *instance variables*. They correspond to the attributes in the class definition. In the `Polygon` class, for example, the instance variables are: `sides`, `angles`, `dimensions`, and `color`. Note that the function `new` returns the written representation of an instance, which consists of the class name and a list of instance variables. An instance variable with a default value is represented as a dotted list, such as `(dimensions . 2)` in our example.

In Gamma, to set or query the instance variable of an instance, the dot notation is used. Thus, each of the instance variables associated with the pentagon instance can now be set as follows:

```
Gamma> pentagon.angles = 108;
108
Gamma> pentagon.sides = 5;
```

```
5
Gamma> pentagon.color = "blue";
"blue"
Gamma> pentagon;
{Polygon (angles . 108) (color . "blue") (dimensions . 2) (sides . 5)}
Gamma>
```

Notice that internally Gamma holds a class instance and its instance variables together in curly braces. This is called *literal instance syntax*.

8.2. Methods

Methods are functions that are directly associated with a class.

We will create a `Lookup` method for the `Catalog` class. This method lets you look up an entry in the catalog by a key associated with the entry. In this example we implement our data as an association list, that is, a list whose elements are also lists, each of which contains exactly two elements : key and value. The library function `assoc_equal` returns the remainder of the association list starting at the element whose key coincides with the key in the argument of the method `Lookup`. Thus, `Lookup` returns the list associated with the key. The special keyword `self` is used when the instance refers to itself within the function:

```
method Catalog.Lookup (key)
{
    car(assoc_equal(key, self.data));
}
```

Note that the keyword `self` can be omitted and the call would look as follows:

```
    car(assoc_equal(key, .data));
```

The calls to class methods are made by instances, using the dot notation. For example, the instance `autoparts` created above can call the `Lookup` method as follows:

```
Gamma> autoparts.Lookup ("muffler");
nil
```

Since the data attribute did not have a default value, the first time call to `Lookup` returns `nil`. In order to put data in the data list, we must create another method:

```
method Catalog.Add (key, value)
{
    local i;

    if (i = .Lookup (key))
    {
        princ("The entry ", key, " already exists\n");
        nil;
    }
    else
    {
        .data = cons(list(key, value), .data);
    }
}
```

Notice that the `Add` method is using `Lookup` to determine whether or not the entry already exists in the association list. If so, it returns `nil`. Otherwise the new entry is added to the data list using the library function `cons`. The return value of a method is the return value of the last function executed within the body of the method.

Now we can add some data. For example, we can add an entry with the keyword "muffler" and the value 1, which is, for example, the number of mufflers in the stock:

```
Gamma> autoparts.Add ("muffler", 1);
(("muffler" 1))
Gamma> autoparts.Add ("starter", 5);
(("starter" 5) ("muffler" 1))
```

Now we can look up the entry for a muffler by the keyword:

```
Gamma> autoparts.Lookup("muffler");
("muffler" 1)
```

Note that the autoparts instance variables can be queried using the dot notation as follows:

```
Gamma> autoparts.data;
(("starter" 5) ("muffler" 1))
```

8.3. Inheritance

Let us consider the following example where a new class is created:

```
class Book Catalog
{
    size =  0;
}
```

The `Book` class is called a *derived* class and the `Catalog` class is called a *base*, or *parent* class for the `Book` class. In addition to having its own attributes, methods, and class variables, a derived class *inherits* all these things from the base class as well. For example, the `Book` class inherits the `data` attribute from the `Catalog` class:

```
(defclass Book Catalog [][data (size . 0)])
```

The relation between the base classes and the derived classes can be described as an "is-a" relation: a derived class "is-a" base class, with its own additional features. Due to inheritance, an instance of a derived class in Gamma can call any method of the base class just like it was an instance of the base class itself:

```
Gamma> math = new(Book);
{Book (data) (size . 0)}
Gamma> math.Lookup("Calculus");
nil
```

In this case it returns `nil` because no entry "Calculus" was added to the list of data. Now we can create an `Add` method for the `Book` class. This method adds an author and a publisher to the association list of data. If the `Add` operation is successful, the size of the list is incremented by 1. This Add method internally calls the `Add` method of the base `Catalog` class using the `call` function. We say that the derived class inherits implementation from the base class. If we were to change the way the `Add` method is implemented in the base class, the implementation would propagate to the derived class.

```
method Book.Add (title, author, publisher)
{
    local pair = list(author, publisher);
    local result = call(self, Catalog, Add, title, pair);

    if (result)
        {
            .size+= 1;
        }
```

```
}
```

The method `Add` can be evaluated as follows:

```
Gamma> math.Add ("Calculus", "Thompson", "Wiley");
1
```

It returns the size of the math catalog as the result of the last evaluated expression within the method. Now if we would like to search for the entry "Calculus", the method `Lookup` is evaluated as follows:

```
Gamma>  math.Lookup ("Calculus");
(Calculus (Thompson Wiley))
```

Classes can be related by "is-a" relations, since one class is a derived class of the other. There can also be "has-a" relations between classes. Let us consider the following example:

```
class Figure
{
    color;
    height;
    width;
}

class Book_1
{
    size;
    figure = new(Figure);
}
```

Class Book_1 "has" an instance of class Figure as an attribute. In other words, class Book_1 *contains* one instance of the class Figure. Let us consider the connections between the methods of the two classes with the "has-a" relations. Suppose the following methods are defined:

```
method Figure.Show()
{
...

}

method Book_1.Show()
{
    .figure.Show();
}
```

The method `Show` of the `Book_1` class internally calls the method `Show` of the `Figure` ("contained") class. We say that the `Book_1` class *delegates* its method to the `Figure` class. Thus, the effect of the delegation is implementation inheritance. It’s true that to inherit implementation, the `Figure` class could be simply derived from the `Book_1` class and the "is-a" relations would be in effect. But then it would be impossible for an instance of the `Book_1` class to have several instances of the Figure class.

Note that in the definition of the `Book_1` class, the `Figure` class is instantiated, which makes the attribute `figure` an *instance* of the class `Figure`. Thus, if an instance of the `Book_1` class is created it can evaluate its `Show` method right away:

```
...
mystery = new(Book_1);
mystery.Show();
...
```

However, if an instance of `figure` is not actually created in a `Book` class definition,

```
class Book_2
{
    size;
    figure;
}
method Book_2.Show()
{
    .figure.Show();
}
```

then it has to be instantiated for each new instance of `Book_2` before any "delegation" will occur:

```
...
mystery = new(Book_2);
mystery.figure = new(Figure);
mystery.Show();
...
```

8.4. Instance Variables

We recall that instance variables (*ivars*), are non-method items that make up an instance of a class. In the example below, the ivars of the instance `math` are `data` and `size`. To set or query the value of an ivar use the class instance and ivar in dot notation:

```
Gamma> math.size;
1
```

Gamma has the ability to add ivars to a class at any time, using the function `class_add_ivar`. As an example consider the `Catalog` class used in the above examples. Suppose we would like to have a variable which holds the date when a catalog is started:

```
Gamma> class_add_ivar(Catalog,#start_date);
nil
Gamma> Catalog;
(defclass Catalog nil [...][data  start_date])
```

Once an instance variable has been added to a class, all new instances of that class created *after* the variable was added will receive the new ivar.

```
Gamma> math;
{Book (data) (size . 0)}
Gamma> cooking = new(Book);
{Book (data) (size . 0) (start_date)}
```

8.5. Class Variables

Class variables (*cvars*), are non-method items that permanently belong to the class in which they are defined. One can think of a class variable as named data associated with the class. There is only ever one copy of the variable. All instances of that class share that copy. All derived classes and all the instances of the derived classes share that one copy. It is like a global variable.

Gamma has the ability to add cvars to a class at any time, using the function `class_add_cvar`. Once a class variable has been added to a class it becomes available to all new instances of that class and the derived classes. However they do not get a private copy of that variable but share one and the same

variable that belongs to the class. As an example consider the `Catalog` class and its derived class, `Book`, once more.

```
Gamma> class_add_cvar(Catalog,#capacity, 200);
200
Gamma> Catalog;
(defclass Catalog nil [ ... (capacity . 200)][data start_date])
Gamma> Book;
(defclass Book Catalog [...][data (size . 0) start_date]
Gamma> history = new(Book);
{Book (data) (size . 0) (start_date)}
```

We can see that the derived class `Book` does not have a private copy of the class variable `capacity`. However this variable is *available* for the derived class as well as for the instances of that class:

```
Gamma> Book.capacity;
200
Gamma> history.capacity;
200
```

To set or query the value of a cvar use the class name (or the instance name) and the cvar in dot notation. Remember, though, a change to the cvar in any class or instance of `Catalog` will change it for all classes and instances of `Catalog`.

```
Gamma> Book.capacity = 300;
300
Gamma> history.capacity;
300
Gamma> history.capacity = 400;
400
Gamma> Book.capacity;
400
Gamma> Catalog.capacity;
400
Gamma>
```

8.6. Constructors and Destructors

A *constructor* is a method that is automatically run when a new instance of a class is made. A *destructor* is a method that is automatically run when the instance is destroyed. Constructors are called for all parent (base) classes of an instance starting with the root of the instance's class hierarchy. Destructors are called for all parent (base) classes of an instance starting with the class of the instance and proceeding toward the root of the instance's class hierarchy.

In Gamma these two methods take the special names `constructor` and `destructor`.

```
method Book.constructor ()
{
    total_books++;
}

method Book.destructor ()
{
    total_books --;
}
```

We'll now set the example variable `total_books` to 2 (since two have already been created: math and history):

```
Gamma> total_books = 2;
2
```

A new `Catalog` object can now be created, and the effect of the constructor and destructor observed:

```
Gamma> biology = new(Book);
{Book (data) (size . 0) (start_date)}
Gamma> total_books;
3
Gamma> biology = nil;
nil;
Gamma> total_books;
3
Gamma> gc();
166
Gamma> total_books;
2
```

The constructor worked as expected, but the destructor appears to have failed. Only after the `gc` function was called did the destructor get called. The `gc` function forces the garbage collector to run. When `biology` was set to `nil` the memory containing the previous definition for `biology` was left unlinked. Once this unlinked memory was recovered by the garbage collector, the destructor was called.

The frequency of the garbage collector running will depend on the program written in Gamma. The garbage collector can be forced to run by using the `gc` function. Occasionally, system activity may prevent it from running immediately, but the requirement to run is noted and it will do so at the next opportunity.

Classes are often used to keep track of real-world objects, and as such, it is important to keep statistics on these objects. One of the most common methods of doing this is by using constructors and destructors to increment and decrement a counter of the number of objects created or currently available.

8.7. Polymorphism

The concept of *polymorphism* has its roots in programming language design and implementation. A language is called polymorphic if functions, procedures and operands are allowed to be of more than one type. In comparison with polymorphic languages, there are languages called monomorphic, such as FORTRAN and C. Being monomorphic means that it is not possible, for example, to define two subroutines in FORTRAN with the same name but different number of parameters.

Overloading is a specific kind of polymorphism which Gamma supports.

8.7.1. Operator Overloading

Operator overloading allows the programmer to define new actions for operators (+,-,*,/, etc.) normally associated only with numbers. For example, the plus operator (+) normally adds only numeric variables. Operator overloading allows the user to define an alternate action to adopt when non-numeric variables are used in conjunction with an operator. The plus operator is often overridden so that strings may be concatenated using the syntax:

```
result = "hello" + " " + "there";
```

When overloading an operator in Gamma the developer must exercise extreme caution since operator overloading is achieved by redefining the operator itself. The typeless quality of variables in Gamma does not allow the interpreter to select an appropriate operator based on the types of the arguments.

Consider the following program fragment:

```
// Assign plus to a function called 'real_plus'.
real_plus = \+;

// Re_define plus to check for strings, and call
// string() or real_plus() depending on arg types.
function \+ (arg1, arg2)
{
    if (string_p(arg1) || string_p(arg2))
        string(arg1,arg2);
    else
        real_plus(arg1,arg2);
}
```

The first step in this example is to re-assign the functionality of the + operator to a function called `real_plus`. Notice that the backslash character is used to pass the + character explicitly. Without the backslash Gamma would interpret the plus character in the function definition statement as a mistake in syntax.

Once this assignment and definition are entered into Gamma the plus operator can be used with strings as well as with numbers:

```
Gamma> 5 + 4;
9
Gamma> "hello" + " " + "there";
"hello there"
```

While it is convenient to set up overloaded functions in Gamma, remember that user functions are generally slower than Gamma's built-in functions.

8.8. Binary Classes and User Classes

Binary classes are classes that are built into the specific version of Gamma that you are using. User classes are those classes defined by the programmer in the process of developing an application.

To test the number of built-in classes in the version of Gamma you are using, start a fresh instance and use the `apropos` function interactively to find all available classes:

```
andrew:/home/andrew > gamma
Gamma;(TM) Advanced Programming Language
Copyright (C) Cogent Real-Time Systems Inc., 1996.  All rights
reserved.
Version 2.4 Build 139 at Jul 6 1999 10:48:51
Gamma> apropos("*",class_p);
(Osinfo)
```

As we can see, **gamma** does not have built-in binary classes. Now let us try to run **phgamma**:

```
andrew:/home/andrew > phgamma
Gamma(TM) Advanced Programming Language
Copyright (C) Cogent Real-Time Systems Inc., 1996.  All rights
reserved.
Version 2.4 Build 139 at July 6 1999 14:21:45
Gamma> apropos("*",class_p);
(Osinfo PhArea PhBlitEvent PhBoundaryEvent PhDim PhDragEvent
```

```
PhDrawEvent PhEvent PhEventRegion PhExtent PhImage PhKeyEvent
PhLpoint PhPoint PhPointerEvent PhPrect PhRect PhRegion PhRgb
PhWindowEvent PtArc PtBarGraph PtBasic PtBasicCallback PtBezier
PtBitmap PtBkgd PtButton PtCallbackInfo PtComboBox
PtComboBoxListCallback PtComboBoxTextCallback PtContainer
PtDivider PtEllipse PtEventData PtFontSel PtGauge PtGenList
PtGenTree PtGraphic PtGrid PtGroup PtHtml PtIcon PtLabel PtLine
PtList PtListCallback PtMenu PtMenuBar PtMenuButton PtMenuLabel
PtMessage PtMeter PtMultiText PtOnOffButton PtPane PtPixel
PtPolygon PtRaw PtRect PtRegion PtScrollArea PtScrollbar
PtScrollbarCallback PtSeparator PtSlider PtTerminal PtText
PtTextCallback PtToggleButton PtTree PtTrend PtTty PtWidget
PtWindow RtTrend)
```

There is a significant difference in supported classes between the **gamma** and **phgamma** executables. The reason is that Photon widgets are mapped into **phgamma** as classes. The standard Gamma executable does not have support for Photon graphics and does not have these built-in binary classes.

User classes are found in the same manner. After user classes are defined they will match the `class_p` predicate in the `apropos` function and be added to the list:

```
andrew:/home/andrew> gamma
Gamma(TM) Advanced Programming Language
Copyright (C) Cogent Real-Time Systems Inc., 1996.  All rights
reserved.
Version 2.4 Build 139 at Jul 6 1999 10:48:51
Gamma> class test
{
    a;
    b;
    c;
}
(defclass test nil [][a b c])
Gamma> apropos("*",class_p);
(Osinfo test)
```

Chapter 9. Tutorial III

9.1. Classes and OOP

Gamma implements object-oriented programming (OOP) features which provide a single-inheritance class mechanism with instance variables and methods. Since Gamma is an interpreter, the object definitions are truly dynamic, allowing for run-time extensibility. This example provides the simplest of starting points to this key software methodology.

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

/*
 *  Demonstrates:
 *  class definitions: attributes and methods.
 *  constructors and destructors
 *  method overloading
 */

/*
 * Define a class of animal, with no default type and a default of 4 legs
 */
class animal
{
    type = "animal";
    num_legs = 4;       // By default, animals have 4 legs
}

/*
 * The constructor for an animal is called when any instance of animal
 * or a subclass of animal is created using a call to 'new'.  Constructors
 * have no arguments.
 */
method animal.constructor()
{
    princ ("A ", class_name(class_of(self)), " is born\n");
}

/*
 * The destructor for an animal is called when any instance of animal
 * or a subclass of animal is deleted by the garbage collector.  There
 * is no explicit deletion mechanism in Gamma.  Destructors have no
 * arguments.
 */
method animal.destructor ()
{
    princ ("A ", class_name(class_of(self)), " dies\n");
}

/*
 * All methods except constructor and destructor are overloaded, meaning
 * that only the method for the nearest class in the ancestry of the
 * instance will be called for any given method name.
 */
method animal.describe ()
{
    princ ("The ", self.type, " has ", self.num_legs, " legs.\n");
}

/*
 * Create a subclass of animal of a particular type.
 */

class cat animal
{
    type = "feline";
```

```
}

/*
 * Create another subclass of animal which is itself a parent class
 */

class insect animal
{
    num_wings = 2;
    num_legs = 6;
}

/*
 * Overload the description method for insects so we hear about
 * wings and legs when we ask about insects.
 */

method insect.describe ()
{
    /*
     * We can explicitly call a method of a parent class using the
     * 'call' function and naming a parent class.
     */
    call (self, #animal, #describe);
    princ (" (oh, and ", self.num_wings, " wings)\n");
}

/*
 * Create a destructor for an insect.  This will be run before the
 * animal destructor.
 */

method insect.destructor ()
{
    princ ("Crunch. ");
}

/*
 * Create a subclass of an insect which is a particular type.
 */

class beetle insect
{
    type = "rhinoceros beetle";
    num_wings = 4;
}

function main ()
{
    local  pet = new (cat);
    local  bug = new (beetle);

    /*
     * cat gets its describe method from the animal class
     */
    pet.describe();

    /*
     * beetle gets its describe method from the insect class
     */
    bug.describe();

    /*
     * Since the destructor will be implicitly called by the garbage
     * collector, we can cause the destructor to occur by removing
     * all references to the instances (set the variables referencing
     * the instances to nil), and then explicitly invoke the garbage
     * collector.  Typically this is not necessary, as the garbage
```

```
     * collector will run when necessary.
     */
    pet = nil;
    bug = nil;

    gc();
}
```

Chapter 10. Interactive Development and Debugging

10.1. Interactive Mode Implementation

The implementation of Gamma's interactive mode provides an interesting example of the how to use the concise power of the language. Interactive mode is implemented with the following few lines of Gamma:

```
princ("Gamma> ");
flush(stdout);
while ((x = read( stdin)) != _eof_)
{
    princ( eval( x));
    terpri();
    princ("Gamma> ");
    flush(stdout);
}
```

An application can easily provide its own customized " interactive mode " by executing a Gamma script file with a variation of this code that is entered when the file is loaded.

10.2. Getting On-Line Help for Functions

Gamma can display function definitions and parameters. To do this, start Gamma interactively and type the name of the function followed by a semicolon and return. For example,

```
andrewt@1:~ >  Gamma
Gamma (TM) Advanced Programming Language
Copyright (C) Cogent Real-Time Systems Inc., 1996. All rights
reserved.
Version 2.4 Build 142 at Aug 25, 1999 21:39:51
Gamma> init_ipc;
(defun init_ipc (my_name &optional my_queue_name domain) ...)
Gamma> new;
(defun new (class) ...)
Gamma> array;
(defun array (&optional &rest contents) ...)
Gamma> insert;
(defun insert (array position_or_function value) ...)
```

Note that the function definitions are described in the internal Lisp representation, as a list. The function is always displayed with the word `defun` first, followed by the name of the function, and then its syntax. The function arguments are enclosed in parentheses, but not separated by commas as they are in Gamma syntax. The Gamma function modifiers (`!`, `?`, and `...`) are represented by: `&noeval`, `&optional`, and `&rest` respectively. For details on these modifiers, see `function` in the Reference section. To give you a general idea, here is how the above functions definitions appear, first in Gamma syntax and then Lisp syntax:

```
init_ipc (my_name, my_queue_name?, domain?)
(defun init_ipc (my_name &optional my_queue_name domain) ...)
new (class)
(defun new (class) ...)
array (s_exp?...)
(defun array (&optional &rest contents) ...)
insert (array, position|compare_function, value)
(defun insert (array position_or_function value) ...)
```

10.3. Examining Variables in a Class or Instance

Classes and instances can be examined in two ways. For a class, you can simply type the name at the prompt. Instances of classes bound to C structures can be viewed using the `instance_vars` function. To examine an instance of a class, simply type an expression which evaluates to that instance (see, for example, `elephant` as an instance of the `animal` class in the Instances section of the Class chapter).

You can examine not only user-defined classes, but also the classes which are implemented in Gamma. For example,

```
Gamma> PhImage;
# < Binary Class: PhImage >
Gamma> instance_vars (PhImage);
[bpl colors flags format image image_tag palette palette_tag size \
 type xscale yscale]
Gamma> x = new (PhImage);
{PhImage (bpl . 0) (colors . 0) (flags . 0) (format . 0) \
 (image . #{}) (image_tag . 0) (palette . []) (palette_tag . 0) \
 (size . {PhDim (h . 0) (w . 0)}) (type . 0) (xscale . 0) \
 (yscale . 0)}
Gamma> x;
{PhImage (bpl . 0) (colors . 0) (flags . 0) (format . 0) \
 (image . #{})(image_tag . 0) (palette . []) (palette_tag . 0) \
 (size . {PhDim (h . 0) (w . 0)}) (type . 0) (xscale . 0) \
 (yscale . 0)}
Gamma> instance_vars (x);
[(bpl . 0) (colors . 0) (flags . 0) (format . 0) (image . #{}) \
 (image_tag . 0) (palette . []) (palette_tag . 0) (size . \
{PhDim (h . 0) (w . 0)}) (type . 0) (xscale . 0) (yscale . 0)]
Gamma> pretty_princ (x, "\n");
{PhImage (bpl . 0) (colors . 0) (flags . 0) (format . 0) (image . #{} ))
  (image_tag . 0) (palette . []) (palette_tag . 0) \
  (size . {PhDim (h . 0) (w . 0)}) (type . 0) (xscale . 0) \
  (yscale . 0)}
t
```

10.4. Using the Debug Prompt

The `debug`> prompt appears when an error occurs in interactive mode. Gamma halts execution of the program and produces the prompt. You can perform any action at the `debug` > prompt that you can perform at the top level, including modifying program source and setting variable values. The value of any variable can be queried by simply typing its name. The calling stack can be queried by using the `stack` function.

The `stack` function displays the execution stack, providing a list with the names of the nested functions executing when the error occurred. The outermost, or top level, function appears first (after `progn`). The function causing the error appears last on the list.

Once the `debug` > prompt appears, the program cannot be continued and must be re-started. If an error occurs again as a result of code executed within the debug level, another nested level of debug will appear. Each level adds to the current point on the execution stack. You can move up debug levels and return to the Gamma> prompt by pressing **`Ctrl - D`** at the `debug`> prompt.

10.5. Debugging a program

The use of an interpreter engine enables some unique approaches to the process of debugging and testing software. This section describes some of the tools and techniques for debugging an application.

10.5.1. Interacting with an Active Program

Gamma can provide an active *view-port* into the running application. Another task (or shell) can, at any time, interact with a running Gamma program, without halting it or otherwise disturbing its real-time response. This provides an approach to debugging that is much more powerful than adding debug print statements.

For example, suppose that we started a process with the name "my_task" interactively:

```
Gamma> init_ipc ("my_task");
t
```

The **gsend** (for Gamma) or **lsend** (for Lisp) utility is used to interact with a running application from a shell:

```
[sh]$ gsend my_task
my_task>
```

The **lsend** utility accepts Lisp input as the default and **gsend** accepts Gamma input as its default.

Once connected to a running Gamma program using **gsend/lsend**, the developer can:

- query/set variables/objects/instance_vars in the global scope
- call functions/methods
- re-define function definitions
- run any Gamma command interactively

The syntax for starting the `gsend` utility is as follows:

gsend [-l] [-g] [*program*] [*pid*]

-l

Accept Lisp input from the keyboard.

-g

Accept Gamma input from the keyboard.

program

a Gamma program name, attached by `name_attach`, `init_ipc`, or `qnx_name_attach`

pid

(QNX 4 only) a task ID

gsend and **lsend** attach to a running Gamma program and allow the user to send commands without exiting the event loop of the attached process. Any statement may be issued, including changing the definitions of existing functions. In our simple example we can call the `princ` function for `my_task` to execute:

```
[sh]$ gsend my_task
my_task> princ ("Hello!\n");
t
```

Notice that event processing stops for the duration of the command. Now let's look at `my_task`. In order to respond to requests from **gsend/lsend**, **my_task** must be executing an event loop. We can start one using the `next_event` function in a `while` statement:

```
Gamma> init_ipc ("my_task");
t
Gamma> while(t) next_event();
Hello!
```

The `my_task` program continues to run as normal during this operation.

This presents an excellent opportunity for rapid development by programmers. Typically developers are used to the " code, compile, link, run, debug, code..." iterative approach to programming. Once a Gamma developer makes a program with a well written event loop, such as the one shown in the section below on trapping errors, programming and testing can become operations that happen in parallel.

Programmers will find that after a piece of code has been written, it can be uploaded to an already running Gamma process with **gsend**/**lsend** by using a simple "cut and paste". The code is automatically assimilated into Gamma and ready to run. Better yet, if there is a problem with the code, the programmer receives immediate feedback and can track the problem down through an interactive debugging prompt that can be built right into the event-loop.

10.5.2. Trapping and Reporting Errors

Gamma provides a pair of functions referred to as the `try`/`catch` mechanism, that is very important for debugging. Consider the following simple event loop:

```
while (t)
{
    next_event();
}
```

This will run until the program exits or an event triggers some code that produces an error condition. There is no protection against errors. Now consider the following setup:

```
while (t)
{
    try
    {
        next_event();
    }
    catch
    {
        princ("error occurred\n");
    }
}
```

This setup of `try`/`catch` will try to evaluate the block of code contained in the "try" portion and jumps to the "catch" portion when an error occurs. A more effective example of the `catch` code block is:

```
while (t)
{
    try
    {
        next_event();
    }
    catch
    {
        princ("internal error: ", _last_error_,
            " calling stack is: ",stack(),"\n");
    }
}
```

This setup provides the developer with information about the last error and the calling stack which led to the last error. Tutorial II provides an example which illustrates the `try`/`catch` and `protect`/`unwind` mechanisms to get reports on an error.

Another setup that the developer may find useful is to automatically start an interactive session in the case of an error. The example of such a setting can be found in Tutorial II.

10.5.3. Determining Error Location

The `stack` function will show the current function calling `stack`, expressed as a list of functions that the interpreter is currently evaluating. To trace the execution path of parts of a program it is useful to print out the code as it is evaluated. The `trace` and `notrace` functions act as delimiters to areas when tracing should occur. The tracing information is delivered to standard output.

The following table of predefined global variables provides additional information useful for debugging:

Table 10-1. Global Variables in Gamma

Global Variable	Description
`_error_stack_`	The stack at the time the last error occured.
`_unwind_stack_`	The stack at the time that an error was discovered.
`_last_error_`	A string containing the last error.

10.5.4. Filtering Object Query Output

Gamma permits the user to control the level of detail reported, and the format used, when an object is queried. This is done by defining a function named `_ivar_filter` with two arguments. For example, each class instance has a number of instance variables that are reported during interactive mode in the format:

```
Gamma> stats = qnx_osinfo(0);
{Osinfo (bootsrc . 72) (cpu . 586) (cpu_speed . 18883) (fpu . 587)
  (freememk . 16328) (machine . "PCI") (max_nodes . 7) (nodename . 2)
  (num_handlers . 64) (num_names . 100) (num_procs . 500)
  (num_sessions . 64) (num_timers . 125) (pidmask . 511) (release . 71)
  (reserve64k . 0) (sflags . 28675) (tick_size . 9999) (timesel . 177)
  (totmemk . 32384) (version . 423)}
```

The following example provides a function named `_ivar_filter` that controls the output format. Note that each instance variable consists of a name and a value. If we define the following:

```
function _ivar_filter (!instance,!value)
{
    princ(format("\n%-20s %-20s", string(car(value)), string(cdr(value))));
    nil;
}
```

then the output for Gamma in interactive mode now looks like:

```
Gamma> stats;
{Osinfo
  bootsrc         72
  cpu             586
  cpu_speed       18883
  fpu             587
  freememk        15544
```

```
machine         PCI
max_nodes       7
nodename        2
num_handlers    64
num_names       100
num_procs       500
num_sessions    64
num_timers      125
pidmask         511
release         71
reserve64k      0
sflags          28675
tick_size       9999
timesel         177
totmemk         32384
version         423
}
```

Chapter 11. Advanced Types and Mechanisms

11.1. Symbols

We introduced symbols in the Symbols and Values section at the beginning of this Guide. Recall that a symbol is a unique word or name within the system, and that references to a particular symbol will be to the exact same Gamma object, regardless of how that reference was obtained. Symbols can be used as variables, as they may have a value that can be queried through evaluation. The value of a symbol can change depending on the current execution scope.

A symbol may contain any character, though it is necessary to escape some characters using a backslash `(\)` when writing them. The normal character set for symbols consists of the following:

- The lowercase letters (a-z)
- The uppercase letters (A-Z)
- The digits (0-9)
- The underscore (_)

A symbol is created by a call to `symbol`, or by reading any legal string of characters which forms a symbol.

11.1.1. Undefined symbols

In Gamma a variable must be assigned a value. A variable does not exist until a value is assigned to it. Once defined, both the value and type of a variable can be changed, effectively re-defining the variable. A variable which is used but has never been assigned a value will cause an error:

```
Gamma> 3 + k;
Symbol is undefined: k
debug 1>
```

The `undefined_p` function can be used to test if a variable is defined, as follows:

```
Gamma> undefined_p (a);
t
Gamma> a = 5;
5
Gamma> undefined_p (a);
nil
```

11.1.2. Uniqueness of Symbols

The uniqueness of symbols in the system provides an interesting way to perform the equivalent of the C language enumerated type, in case you want a list of constant values representing different things. In Gamma, when two symbols are tested for equality, the comparison is first done on the symbol reference itself, not the value associated with the symbol. This is because the Gamma `==` operator is mapped to the Lisp `equal` function, which determines equality first with the `eq` function. The Lisp `eq` function tests for equality of the reference, and only if this has failed will the equal function perform an equality test on the value of the references. Also in Gamma, a symbol can be defined without assigning a value.

When the Gamma engine reads a literal symbol (see Literal Syntax and Evaluation in this chapter), as illustrated in the example below, it determines that the reference is in fact a symbol. If the symbol does not exist, Gamma creates the it with a value of `_undefined_`.

```
Gamma> x = #yes;
```

```
yes
Gamma> yes;
Symbol is undefined: yes
debug 1>
```

Therefore, it is valid in Gamma to make comparisons for equality between symbols whose values are not defined. Such comparisons between symbols are actually more efficient than comparing the values of two symbols. This leads to the following example. Here Gamma uses the equivalent of an enumerated type, but it is more efficient than assigning actual values to the constants, since the test is for the symbol reference only.

```
Gamma> function my_state (x)
{
if (x==#on)
    princ ("I am on.\n");
else if (x==#off)
    princ ("I am off.\n");
else if (x==#unstable)
    princ ("I am not stable.\n");
else
    princ ("I don't know.\n");
}
(defun .......)
Gamma> a = #on;
on
Gamma> my_state (a);
I am on.
t
Gamma> my_state (#off);
I am off.
t
```

Notice that the enumerated set (on, off, unstable) was not created as variables with assigned values (1, 2, 3,...) but used directly, leading to a more efficient and cleaner implementation.

11.1.3. Properties

Symbols can be assigned properties, using the `setprop` function. Each assigned property is a name/value pair, which is globally defined for the symbol, regardless of the current scope and value. These properties can be accessed using the `getprop` function.

11.1.4. Predefined Symbols

There are some symbols, such as `_undefined_` mentioned above, whose values are predefined in Gamma. For the complete listing of symbols that are predefined in Gamma see section Predefined Symbols in the Reference Manual.

11.2. Evaluation

Expressions in Gamma consist of a few basic types, which are submitted to an evaluator to produce a return value. Every evaluation returns a result. Any Gamma object may be submitted to the evaluator. The evaluator behaves differently according to the type of the evaluated object.

Table 11-1. Type Evaluation

Type of Object	Evaluation result
symbol	The value of the symbol in the current scope.
list	Function or method call.
all others	Itself.

11.2.1. Evaluation of a Symbol

If a symbol is being used as a variable, then its value will depend on the scope in which it is being evaluated. A new scope is entered whenever a user-defined function is executed, or when a `with` statement is executed. A symbol is defined within a scope when it appears in the argument list of a function, or when it appears in the variable list of a `local` statement. If a symbol is not defined within the current scope, then the value of the symbol in the next most local scope is used. This is called *dynamic* scoping, since the scope of a symbol used in a function may depend on the calling sequence that executed that function. The value of a symbol may be any Gamma object. See examples on dynamic scoping in Tutorial II.

11.2.2. Evaluation of a List

Since Gamma is built on top of Lisp, all Gamma statements and expressions are translated into lists, which represent the equivalent function call. For example, the expression `2 + 3;` is translated into the function call `(+ 2 3)` at read time, and evaluated as a list at run time. (Operators like `+` are functions in Lisp.) For more details on Lisp function syntax, see Getting On_Line Help for Functions.

When a list is submitted to the evaluator, it will be treated as either a function call or a method call. The first element in the list is evaluated, and the result examined. If the first element evaluates to a function definition, then the remainder of the list is treated as the arguments to this function, and the function is called. If the first element evaluates to an instance of a class, then the second element is evaluated.

If the second element evaluates to a class, then the third argument must be a symbol that names a method for that class. If the second element does not evaluate to a class, then it must be a symbol that names a method for the class of the instance, or a method of one of the instance's parent classes.

Once a method has been identified, the remaining elements of the list are treated as the arguments to the method, and the method is called on the instance. For example, here we call a function with the given arguments:

```
(function arg1 arg2...)
```

Here we call a method from an instance's own class.

```
(instance method_name arg1 arg2...)
```

And here we call a method from the instance's class hierarchy.

```
(instance class method_name arg1...)
```

If an explicit class is provided, it is normally a parent (base) class of the given instance. This is not enforced by the evaluator, so it is possible to call a method for an instance which is not a member of the class for which the method was defined. This is not normally a good idea, and can be highly confusing to anybody reading the code.

11.2.3. Evaluation to Itself

Most Gamma object types evaluate to themselves, meaning that the result of submitting the object to the evaluator is the object which was submitted, unmodified. Any object except a list or a symbol will simply return itself when evaluated. For example, the number `5`, when evaluated, will return `5`, which is itself.

11.3. Literal Syntax and Evaluation

One of the most powerful features of an interpreter-based language such as Gamma is the ability to evaluate symbols and expressions at run-time. Gamma uses the # operator to indicate a literal expression. For those familiar with Lisp, this is equivalent to the forward quote syntax. Gamma also supports evaluation of sub-expressions, using the `‘` and `@` operators. For more details on their use, see Quote Operators and further explanation below.

11.3.1. Literal Expressions

A literal expression is the expression that specifies an actual value rather than a function for creating the value. For example, the number `3` is a literal, where the expression `(+ 1 2)` is not. Similarly, the string `"hello there"` is a literal, and the expression `string("hello ", "there")` is not, yet they produce equal values when evaluated.

Most object types in Gamma have Lisp and Gamma literal forms. You can create a valid object of some types (such as numbers, symbols, and strings) by reading a literal from a file or the command line. Other types (such as arrays, classes, and functions) are created by corresponding statements or functions.

See Literals in the Reference Manual for definitions, notations, and examples of literal expressions in Gamma.

11.3.2. Deferring Expression Evaluation

A Gamma expression preceded by the quote operator (#) will be taken literally, i.e., it will be protected from the evaluator. When the symbol containing this literal is evaluated, its contents are then interpreted. For example:

```
Gamma> x = #5 + 6;
(+ 5 6)
Gamma> #x;
x
Gamma> x;
(+ 5 6)
Gamma> eval (x);
11
```

In the first case, the quote operator (#) protects the entire expression from the evaluator. That is, it protects everything to its right, all the way to the end of the expression (usually a semicolon or closed parenthesis). In the second case it is used to "produce" the literal symbol `x`. Then `x` is evaluated, returning its literal contents. Finally, the `eval` function is used to force execution of the literal contents of `x`. The `eval` function forces the resolution of variable references, as in this example:

```
Gamma> a = 1;
1;
Gamma> x = a + 5;
6;
Gamma> x = #a + 5;
(+ a  5)
Gamma> a = 10;
10
Gamma> eval (x);
```

```
15
```

The literal is often used to delay the evaluation of an expression until an event is triggered. A good example is the `add_set_function`. This function takes two arguments. The first argument must be a symbol, so the # operator is used to prevent the required symbol from being evaluated. The second argument is simply any expression, most commonly a function. The `add_set_function` function sets the second argument to be evaluated when the first argument is changed:

```
Gamma> add_set_function (#a, #princ("My value = " ));
(princ "My value =")
Gamma> a = 21 / 3;
My value = 7
```

In the following variation of the above example, a symbol used as an argument has been assigned a literal symbol, so that its evaluation will result in the desired symbol:

```
Gamma> x = #b;
b
Gamma> add_set_function (x, #princ("My value = " ));
(princ "My value =")
Gamma> b = 21 / 3;
My value = 7
```

11.3.3. Literal Function Arguments

Some functions require arguments that are symbols. Normally, the arguments to a function are evaluated before the function is actually invoked. It is possible to cause a function's arguments not to be evaluated by using the *exclamation* modifier in the function declaration.

As an example, we can write our own version of the `add_set_function` function mentioned above:

```
Gamma> function my_add_set (!sym, !exp)
            {add_set_function(sym, exp);}
(defun my_add_set (&noeval sym &noeval exp) (add_set_function sym exp))
Gamma> my_add_set (c, princ("My value = "));
(princ "My value = ")
Gamma> c = 21 / 3;
My value = 7
```

For more details on function arguments see Function Arguments in the Functions and Program chapter of this Guide.

11.3.4. Partially Evaluated Literal

Gamma supports evaluation of sub-expressions, allowing you to write expressions whose elements may or may not be evaluated. In the following example, we want to create a literal expression which will calculate `a` to the power of `b`, where `b` is a specific power, evaluated at the time the literal is defined. The operator `‘` is used like # to prevent evaluation of the expression, but it allows for exceptions. These exceptions, which will be evaluated, are denoted using the `@` operator.

```
Gamma> b = 3;
3
Gamma> my_cube = ‘(pow (a, @b));
(pow a 3)
Gamma> a = 10;
10
```

```
Gamma> eval(my_cube);
1000
```

In the following example, the timer event is used to demonstrate how the current value of a variable can be evaluated into a literal:

```
Gamma> a = "hello";
"hello"
Gamma> every (15, #princ(a,"\n"));
1
Gamma> next_event();
hello
nil
Gamma> a = "goodbye";
"goodbye"
Gamma> next_event();
goodbye
Gamma> cancel (1);
[866239787 836823463 15 ((princ a,"\n")) 1]
Gamma> every (15, list (#princ, a, "\n"));
2
Gamma> next_event();
goodbye
nil
Gamma> a = "no more";
"no more"
Gamma> next_event();
goodbye
nil
```

11.3.5. Constructing Variable Names at Run-time

Controlling when an expression is evaluated lets you generate the actual variable names at run-time. This can produce extremely concise code, particularly compared to the C language equivalent. In the following example, a set of simple objects each has a value. The object name and its value is entered. In a conventional language, we might search the array of objects to find the one with the given name, and then make the assignment. Gamma makes it possible to directly construct the variable reference using the `set` function, as follows:

```
Gamma> name = "fido";
"fido"
Gamma> value = "bites";
"bites"
Gamma> set(symbol(string(name)), value);
"bites"
Gamma> fido;
"bites"
```

Note that the syntax does not accept the = assignment operator, so the functional form of the assignment operator: `set` must be used. Note also that we would probably use `undefined_p` to verify that the variable actually existed to avoid halting the program due to an undefined variable error. Although the example is trivial, this technique is very useful for constructing function references based on run-time data.

11.3.6. Literal Array Syntax

An array is defined in Gamma with the `array` function, which creates the array and sets the elements to the specified values.

Gamma uses the familiar square brackets `[ ]` syntax to reference array elements. Although Gamma has the functions `aref` and `aset` for reading and writing specified elements of the array, the square bracket syntax is normally used. An array is automatically re-sized if an element beyond its current size is set. Arrays do not have a type, and array elements can be of different types. Array elements can be set to literals (including expressions protected from evaluation), as in the following example:

```
Gamma> x = array (3, "hi");
[3 "hi"]
Gamma> x[3] = #a + 5;
(+ a 5)
Gamma> x;
[3 "hi" nil (+ a 5)]
Gamma> a = 8;
8;
Gamma> eval(x[3]);
13
```

Generally, literal arrays should be avoided except for static variables. A literal array is embedded into your code. If it is changed, then the code is effectively changed!

Chapter 12. Input and Output

12.1. Referencing Files

As in C, there are two ways to reference a file in Gamma, using a *descriptor* or a *pointer*.

A file descriptor is an integer that identifies an open file within a process. This is the lowest-level handle available for interacting with the open file. Disk files, pseudo-ttys, IP sockets, UNIX-domain sockets, pipes and other facilities all offer interaction through file descriptors.

A file pointer is an abstraction of the file that adds buffering on input and output. This would be of type FILE* in C. The reason this exists is that it is very inefficient to use a file descriptor, which does not perform any in-process buffering where many reads and writes are being performed. The file pointer stores many write requests until it has enough data to perform a more efficient write to disk, hopefully in multiples of the disk block size.

In Gamma, a file pointer is an opaque structure (the internals are not visible to the programmer) that is effectively a buffered file. (See the note in `open`.) It's abstracted a little further to also include strings as file pointers, when they are opened using `open_string`.

Some Gamma I/O functions work with file descriptors (generally those that start with `fd_`), others work with file pointers, and a few work with both.

12.2. Lisp and Gamma I/O mechanisms

Gamma provides sophisticated mechanisms for reading and writing expressions, which can greatly simplify most file manipulation functions. There are two fundamental facilities for manipulating file data in Gamma: the reader and the writer. Gamma is based on the SCADALisp engine, and acts as a read-time translator from Gamma syntax to SCADALisp internal form. Thus expressions can be read in either Gamma or Lisp (we abbreviate SCADALisp as simply Lisp) representation.

Since the internal representation of an expression is an optimized Lisp mapping, the writer will produce its output as Lisp. This makes the reader and the writer symmetrical in Lisp, but not symmetrical in Gamma. A purely symmetrical Gamma writer is not possible, since there is no way to express literals in Gamma for data types such as list, buffer, array, instance and class.

The Lisp writer is aware of the format that the Lisp reader requires, and is able to format any expression such that the reader can subsequently read it back in. This means that an arbitrarily complex expression, such as a list containing instances of a class whose instance variables include arrays and other instances, can be written using a single line of code, and read back in using a single line of code as well. Since a Gamma function is simply a data object, it can be written and read in exactly the same way.

Generally, the lack of symmetry between the Gamma reader and the Lisp writer is not a problem, since any data written by Gamma will still be readable simply by instructing the `open` function to recognize Lisp instead of Gamma syntax.

12.3. Writing

12.3.1. Print vs. Princ

It is not always appropriate to write a data item in a way that can be read by the Lisp reader. For example, the Lisp reader requires that all character strings are surrounded by double quotes to differentiate them from symbols and to deal with white space and special characters. In some cases, the programmer may

wish to write a character string in "human-readable" form, with no quotes and escapes on special characters.

The Gamma writer will produce both kinds of output. The `print` function will always generate output which can be read by the Lisp reader, including escape characters, quotation marks and buffer and instance special forms. The `princ` function attempts to make the output as readable as possible to a human, but will not necessarily produce output that can be read by the Gamma reader. The name `princ` is historical, and can simply be thought of as an alternate form of `print`. Notice that neither `princ` nor `print` will automatically place a carriage return at the end of a line. The programmer must explicitly print a "\n" or make a call to `terpri`.

12.3.2. Write vs. Writec

Like `princ` and `print`, there are two forms of the `write` function. The `write` function operates identically to the `print` function, except that its first argument declares the file handle to which it will write its output. The result of a `write` function is machine readable, whereas the result of a `writec` function is intended to be human readable. Notice that neither `writec` nor `write` will automatically place a carriage return at the end of a line. The programmer must explicitly print a "\n" or make a call to terpri.

12.3.3. Terpri

The `terpri` function will produce a carriage return either to the standard output or to a given file handle. `terpri` is most commonly used to generate a carriage return in a file that is being written using the `write` function. If the programmer were to use (`write file "\n"`) then the file would actually contain the four characters "\n", rather than the intended carriage return. `terpri` will insert a carriage return into the file under any circumstances.

12.3.4. Pretty Printing

All of the printing functions have a further variant, known as the pretty printing functions. These variants attempt to format the output to an 80-column page, inserting line breaks and white space in order to make the output more readable. The pretty-printing indentation rules are intended to make data structure and program flow more easily understood, and closely follow the pattern used by GNU Emacs in its Lisp indentation mode.

12.3.5. Printing Circular References

It is common when programming with dynamic lists and arrays, or when constructing inter-related class definitions, to create a data structure which is self-referential, or which contains circular references. For example, it may be useful to have a child class contain a pointer to its parent, and the parent class contain a pointer to its child. In this circumstance, an attempt to print an instance of the child class would cause the Lisp writer to enter an infinite loop if it did not take precautions. In C programs, this circumstance is normally avoided by having a printing routine which understands the child/parent relationship and simply writes them in such a way that the infinite loop is never entered. This carries the problem that each data structure must have its own dedicated printing routine, which necessarily does not preserve a generalized data syntax, and which cannot perfectly represent the child/parent relationship in any but the simplest of cases.

Gamma solves the problems of self-reference and circularity by modifying the printed representation of an object to include embedded reference points in the data structure. Whenever a Gamma object is printed, all circular references and self-references are detected before the object is printed, and reference points are inserted into the printed representation. Subsequent attempts to print an object that was

previously printed will merely produce a reference to the first printing of the object. This facility produces a result that is essentially impossible in languages such as C; it perfectly preserves multiple pointer references to data which are not known, a priori, to be multiple references.

A very simple example of self-reference may be a list that contains itself. This is normally achieved using destructive functions such as `nappend`, `rplaca` and `rplacd`. Consider the following dialogue:

```
Gamma> a = list(1, 2, 3);
(1 2 3)
Gamma> rplacd (cdr (a), a);
(1 2 (1 2 (1 2 (1 2 (1 2 (1 2 (1 2 (1 2 (1 2 ...)))))))))
```

In this case, by replacing the tail of the list with the list itself, it is possible to create a self-referential list which cannot be printed using normal means. Any attempt to print this list will cause an infinite loop. The Lisp writer in fact produces the following output:

```
Gamma> a = list(1, 2, 3);
(1 2 3)
Gamma> rplacd (cdr (a), a);
#0=(1 2 . #0#)
```

The first time that the self-referential list is printed, the Gamma writer determines that a self- reference will occur, and marks that point with a numbered place holder, using the syntax #n=, where n is a monotonically increasing number counting the number of circular references in the data object. Each subsequent reference to the marked object will cause the writer to produce a reference back to the original using the syntax #n#. For example, if we create another, similar list, and then put both lists together into another list, we will get the following:

```
Gamma> b = list(4, 5, 6);
(4 5 6)
Gamma> rplacd (cdr (b), b);
#0=(4 5 . #0#)
Gamma> d = list(a,b);
(#0=(1 2 . #0#) #1=(4 5 . #1#))
```

Using this method, arbitrarily complex objects can be written, with all circular and self-references maintained.

As a side effect of this printing mechanism, duplicate references to objects which are not circularly defined will also be caught and correctly reproduced. For example, suppose that a list contains a single string more than once. It would be wasteful to write that string many times, and would generate an incorrect result on reading if the multiple references to that string were not preserved. The Lisp writer will correctly handle this situation:

```
Gamma> x = "Hello";
"Hello"
Gamma> a = list (x, x, x);
(#0="Hello" #0# #0#)
Gamma> eq (car(a), cadr (a));
t
```

In the above example, if the Gamma writer did not preserve the multiple references to the string "Hello", then `a` would be printed as:

("Hello" "Hello" "Hello")

When this object is read by the Gamma reader, we would get a list which is visibly the same but for which the data references no longer match:

```
Gamma> a = list("Hello", "Hello", "Hello");
("Hello" "Hello" "Hello")
Gamma> eq (car(a),cadr(a));
nil
```

12.4. Reading

12.4.1. Reading Gamma Expressions

Any valid Gamma expression can be read by the Gamma reader using the function `read`. The `read` function will read from the current location in a file, skipping over comments, until it encounters a character which could be the beginning of a Gamma expression. The reader then constructs the shortest possible complete expression from the input and returns that. A complete Gamma expression may be as simple as a number, or as complex as a complete function definition or complex data object. The reader ignores white space, except as a token separator. It may be interesting to note that the entire Gamma mainline is essentially just a simple loop:

```
while ((exp = read (input_file)) != _eof_) eval (exp);
```

12.4.2. Reading Arbitrary ASCII Data

Gamma allows the programmer to read arbitrary ASCII data using the function `read_line`, which will read from the current file position to the first carriage return, regardless of the syntactic validity of the data on the line. If data fields are known to be separated by white space, then the `read` function using Lisp syntax may also be used to read a single field. Notice that the `read` function will treat an unquoted string of ASCII characters as a symbol, not as a string. It is more common when dealing with line-formatted data to use `read_line` followed by `string_split`.

12.4.3. Reading Binary Data

Gamma provides a number of functions for reading binary data. These functions all begin with the prefix `read_`, and they read according to the rules for C data types for the particular platform. For example, `read_char` will read a decimal representation of a string of length 1 containing a single character.

Chapter 13. Special Topics

13.1. Modifying QNX Process Environment Variables

The QNX 4 environment variables can be read and modified by a Gamma program using the `getenv` and `setenv` function calls.

13.2. QNX 4 Interprocess Communication (IPC)

QNX 4 interprocess communication is a popular mechanism for 'talking' between software modules, based on the QNX 4 operating system. QNX 4's microkernel architecture implements message passing in such a way that only data locations are transferred between processes, making its IPC for small amounts of data as, or more, efficient than shared memory schemes. Gamma encapsulates most of the common QNX 4 IPC 'C' function calls.

Functions such as `qnx_receive` and `qnx_reply` may be redundant in a program that is using one of Gamma's built-in event-loop mechanisms. To review the built-in functionality of Gamma's event loops refer to the section on event loops.

The `qnx_name_attach` function attaches a 'name' to the current process. Names, rather than PIDs, are convenient ways to look for tasks since they are static while the PID of a program will not be.

Names are ASCII strings up to 32 character in length and can be either local or global. Local names must be unique to the node. Any attempt to register an existing local name will fail. Global names allow duplication and start with a slash '/' character. Global names are stored within a name program in QNX 4 called *nameloc*. When one process wants to look up another process's name, the `qnx_name_locate` function is called and the name to PID mapping is completed.

```
Gamma> myname = qnx_name_attach(0,"my_app");
20
```

The first argument to the `qnx_name_attach` function is the node on which to register the name. If the node number is zero the local node is assumed. `qnx_name_attach` returns a name id which is used with the `qnx_name_detach` function.

The `qnx_name_detach` function removes a local or global name from the local name list or the DataHub.

```
Gamma> qnx_name_detach(0,myname);
t
```

As with the `qnx_name_attach` function, the first argument to the `qnx_name_detach` function is the node number. The second argument must be the name id returned when attaching the name.

Once a name is registered then the `qnx_name_locate` function is useful for locating the task by name. The return value of this function is a dotted list of the format: `(pid . copies)` The pid is the process ID of the located task and copies is the number of processes that matched the name. Local names must be unique but there can be multiple instances of global names (those starting with '/').

An example of using the `qnx_name_locate` function follows:

```
Gamma> queue = qnx_name_locate(0,"qserve",0);
(91 . 1)
```

The PID of the `qserve` task is 91 and there was a single instance of that registered name found.It is important to assign the return value of the `qnx_name_locate` function to a variable since it is the first

number in the list (PID) that is used as an argument to Gamma functions such as `qnx_send`, `qnx_receive`, `qnx_reply`, `qnx_vc_attach`, and `qnx_vc_detach`.

The `qnx_receive` function allows for the Gamma engine to remain receive-blocked on a specific PID, waiting for a message.

IMPORTANT: If Gamma is being run using a built-in event loop or using the `next_event` or `next_event_nb` functions then using the `qnx_receive` function MAY BE REDUNDANT. Event loops in Gamma have a built-in receive/reply mechanism.

The `qnx_send` function uses the QNX 4 `send` C/C++ function to send information between tasks. The `qnx_send` function is a synchronous IPC function, and as such, the sending task waits for the receiving task's reply before continuing.

The `qnx_send` function can be used to send Gamma expressions between Gamma modules. Gamma ships with a number of example programs, of which example 12 demonstrates the use of `qnx_send` to transmit and execute a function on another module. (Examples can be found in /usr/cogent/examples/ directory)

The important excerpts from this example are:

```
task = car (qnx_name_locate (0, "gui", 1000));
qnx_send (task, stringc (#Arc.fill_color = PgRGB(0xff, 0xff, 0)));
function TitleClock()
{
    win.title = date();
}
// Transmit new function
qnx_send (task, stringc (TitleClock));
// Execute new function once.
qnx_send (task, stringc (#TitleClock()));
```

The communications channel is opened by locating the task using the `qnx_name_locate` function and then using `qnx_send`. The first `qnx_send` sends a command for the receiving task to evaluate, in this case to change the fill color of an object named 'Arc'. The `stringc` function is used to produce an expression that is parse-able.

Next, a new function is defined, passed, and executed on the other task using two separate `qnx_send`'s.

If you are sending IPC messages to a non-Cogent IPC task the `send_string` and `send_string_async` functions should be used.

13.3. Cogent IPC

The Cogent IPC layer is a generalization of QNX 4's send/receive/reply IPC layer. Cogent IPC has many benefits that allow users to easily code what would be complex systems in C. Some of these services are:

- Network-wide name registration service
- Cogent DataHub exceptions and echos
- true asynchronous messages
- pseudo asynchronous messages
- synchronous messages
- QNX 4 IPC messages
- Task started notification
- Task death notification
- automatic handling of receive/reply for Cogent IPC messages

- remote procedure calls

13.3.1. Cogent IPC Service Modules

To use the Cogent IPC layer, two services optionally provided to the Gamma developer are required: **nserve** and **qserve**. These services are run as programs on the same CPU or network as Gamma.

The **nserve** command is the Cascade NameServer module. Although similar to the QNX 4 **nameloc** program in concept, this name database has some differences that make it worth using.

The **nserve** module is run on every node requiring name services. Every **nserve** module is updated on an event-basis, rather than on a timed basis as QNX 4's **nameloc** is, and therefore discrepancies between multiple nserve's on a network are rare.

The **qserve** program is the asynchronous queue manager for Cogent IPC;. Queues are used in Cogent IPC to implement asynchronous communication channels between two programs. The **qserve** module is run on every node requiring Cogent queue services.

13.3.2. Cogent IPC Advanced services

The Cogent IPC layer provides many advanced services that augment the basic send/receive/reply protocol. This section describes those services.

13.3.2.1. Cogent IPC Messages

The Cogent IPC layer provides a messaging protocol that is easier to use and different in format from raw QNX 4 send/receive/reply.

Messages between Cogent IPC-enabled tasks are very similar to function calls. A message is constructed and sent, and the task on the other end evaluates the message. The return value of the evaluation of the message is transmitted to the originating task in the reply.

Consider two Gamma modules using the following code:

Task A:

```
#!/usr/cogent/bin/gamma
init_ipc("task_a");

while (t)
{
    next_event();
}
```

The function `init_ipc` is called first to initialize Cogent interprocess communication. For more details, see IPC Initialization below.

Task B:

```
#!/usr/cogent/bin/gamma
init_ipc("task_b");

function ask_taska_date ()
{
    local result,tp;
    if (tp = locate_task("task_a",nil))
        result = send(tp,#date());
    else
        result = "could not locate task A";
}

every(1.0,#princ(ask_taska_date(),"\n"));
```

```
    while (t)
    {
        next_event();
    }
```

Of specific note in this example is the format of the message in the `send` function. The first argument to the Cogent IPC function `send` is a task. The `locate_task` function, along with the `nserve` module provides the name lookup. The second argument is an expression for the receiver to evaluate. For simple `send`'s an unevaluated Gamma expression (using #) will suffice. For more complex `send`'s, such as when a partially evaluated list of arguments need to be passed, the format of the send command should be Lisp.

This code gives a good example of using the Cogent IPC layer as an RPC (Remote Procedure Call) mechanism.

To use the Cogent IPC layer for transferring data between tasks, use the Lisp expression for assignment: `setq`. An example is:

Task C:

```
    #!/usr/cogent/bin/gamma
    init_ipc("task_c");

    add_set_function(#x,#princ("Task C reports x=",x,"\n"));

    while (t)
    {
        next_event();
    }
```

Task D:

```
    #!/usr/cogent/bin/gamma
    init_ipc("task_d");

    function inc_x ()
    {
        local result,tp;
        x++;
        if (tp = locate_task("task_c",nil))
            result = send(tp,list(#setq, #x, x));
    }

    x = 0;
    every(0.1,#inc_x());

    while (t)
    {
        next_event();
    }
```

In this example task C sets up a `set_function` before starting its event loop. The set function will print out the value of x if it changes. Task D initializes x to 0 and then starts a timer to run every tenth of a second to increment x and send `setq` expressions to task C.

```
    (setq x 1)
(setq x 2)
(setq x 3)
(setq x 4)
```

These expressions are in Lisp format because all messages between processes use the Lisp internal representation for efficiency.

The `setq` function is evaluated in task C. Any side effects of the function, for example the setting of the variable x, happens in task C. The return value of the function is the content of the reply message. The return value of the `send` function can be found by evaluating the 'result' variable in the `inc_x` function.

Consider the `inc_x` function re-written as:

```
function inc_x ()
{
    local result,tp;

    x++;

    if (tp = locate_task("task_c",nil))
    {
        result = send(tp,list(#setq, #x, x));
        princ("task D result of send: ",result,"\n");
    }
}
```

When this example is run the return value of the send is shown to be the result of the `setq` function. Obviously, task D must wait for task C to receive and evaluate the message before sending back the response.

13.3.2.2. Asynchronous Messages

Consider two tasks that wish to communicate: task E and task F. Task E is a time sensitive task that needs to deliver a package of data to task F. Task E cannot take the chance that task F will accept its data immediately and issue a reply so that it may continue with its own jobs. In short, a synchronous send compromises task E's job because it must wait for task F to respond before proceeding.

To send data asynchronously from task E to task F, a queue is used. Data is sent from task E to the queue. The queue responds immediately to task E, freeing it up to continue. Then a *proxy*, a special non-blocking message, is sent from the queue to task F. Upon receipt of the proxy, task F knows that the queue contains data for it. When task F is ready it asks the queue for the data.

With some small changes, the example from the previous section can be changed from synchronous messaging to asynchronous, as follows:

Task E:

```
#!/usr/cogent/bin/gamma
init_ipc("task_e","task_e_q");

add_set_function(#x,#princ("Task E reports x=",x,"\n"));

while (t)
{
    next_event();
}
```

Task F:

```
#!/usr/cogent/bin/gamma
init_ipc("task_f","task_f_q");

function inc_x ()
{
    local result,tp;

    x++;
```

```
    if (tp = locate_task("task_e",nil))
    {
        result = send_async(tp,list(#setq, #x, x));
        princ("task F result of send: ",result,"\n");
    }
}

x = 0;
every(0.1,#inc_x());

while (t)
{
    next_event();
}
```

The `init_ipc` function calls at the beginning of each module now open a queue name with **qserve**, and the `inc_x` function has been changed to use `send_async` instead of `send`.

When this example is run the results show that task `F` receives a `t` (true) that the message was delivered but does not have to wait for task `E` to generate the result of the expression.

Using asynchronous communication immediately solves the dead-lock problem that all developers of multi-module systems must eventually face. To the developer, the use of asynchronous communication in Gamma entails only the use of a slightly different function: `send_async` instead of `send`.

13.3.2.3. Pseudo-Asynchronous Messages

For situations where the **qserve** program is not running and an asynchronous non-blocking IPC call is required then Gamma pseudo-asynchronous IPC call can be used.

The `isend` function sends a message between two Cogent IPC enabled tasks. Immediately upon receipt of the message, the receiver replies that the message was received. The return value of the received message is not sent back.

13.3.2.4. Task Started & Death Notification

When a task registers a name with nserve it can thereafter receive information regarding any other nserve registered task that starts or stops.

This is done by defining two functions with specific names, each within their respective code, to handle this information. The functions are:

```
function taskstarted_hook (name, queue, domain, node, id);
```

and

```
function taskdied_hook (name, queue, domain, node, id);
```

The body of each of these functions is up to the programmer. Most "hook" functions check the name, queue, and possibly the domain of the started/stopped task and then take a specific action such as:

- restarting a task that has died;
- informing the user that a module has died;
- inform other modules that a new service is available;
- query the new module for information; and,
- Cogent DataHub start/stop.

13.3.2.5. Automatic Handling of QNX 4 receive and reply

The following Gamma functions automatically handle QNX 4 `receive`/`reply`:

- `PtMainLoop`
- `next_event`
- `next_event_nb`
- `flush_events`

13.3.2.6. IPC Initialization

Before any form of Cogent interprocess communication occurs there must be a call to the `init_ipc` function. This function opens the channels of communications between Gamma and other tasks powered by Gamma, Cascade Connect, or other Cogent products. With this function you determine your task's name and optionally its queue name and domain.

A program's name is the string registered with the nserve program. Gamma names and queue names for tasks should be unique on the network. A program's queue name is the name of the queue that is registered if it wants to participate in asynchronous communication using Cogent's **qserve** utilities. The domain name is the name of the default Cogent DataHub domain from which to read and write points.

It is typical to find the `init_ipc` function called within the first few calls in the program. Here's an example:

```
#!/usr/local/bin/gamma
require_lisp("PhotonWidget");
require_lisp("PhabTemplate");

myname = car(argv);
init_ipc("myname");
```

This program segment first defines the engine to run on the first line, then loads some required files for Photon widget manipulation and Photon Application Builder support. The `argv` variable holds the arguments passed to Gamma. The first item in the list is the name of the executable, which is put in the myname variable. The `init_ipc` function is then called with the registered name being whatever the name of the program happens to be.

13.3.2.7. Locating Tasks

Using Gamma's IPC communications protocol, a task can be located by name or by id. This protocol allows for synchronous, asynchronous, and semi-asynchronous communications between Gamma, SCADALisp, and other Cogent products such as Cascade Connect and the Cogent DataHub.

Locating a task by name can be done with the `locate_task` function. This is similar to using the `qnx_name_locate` function except that, since nserve's names are intended to be unique on a network the node number need not be specified.

```
marko:/home/marko$ gamma -q
Gamma> init_ipc("locate_test");
t
Gamma> tp = locate_task("cadsim",nil);
#< Task:13424 >
```

The return value of the `locate_task` function is a Gamma task type. The task type is an internal representation of the task that was located. There is nothing the user can do with variables of this data type other than to pass them through as arguments to Cogent IPC functions.

To locate a task on a specific node with a specific PID number use the `locate_task_id` function.

Before using either `locate_task` or `locate_task_id`, the `init_ipc` function must have already been called.

Once discussions with a task are completed, the channel should be closed using the `close_task` function.

13.3.2.8. Transmitting Character Strings

The `send_string` and `send_string_async` functions are used to format a message to be sent to a non-Cogent IPC task. These functions will accept a string (text surrounded by quotes) as a parameter, and will send the contents of the string without the enclosing quotes. Note that the normal `send` function will send the enclosing quotes as part of the message.

13.3.3. Cogent DataHub

The Cogent DataHub is a high performance data collection and distribution center designed for easy integration with a Gamma application. Just as QNX 4 is an excellent choice for developers of systems that must acquire real-time data, the Cogent DataHub is the right choice for distribution of that data.

The Cogent DataHub provides:

- data services to its clients by exception and lookup;
- asynchronous data delivery ensuring client task protection blocking;
- network connection/reconnection issues;
- data services to many clients at once;
- transparent data services to/from Gamma;
- flexible data tag names;
- inherent understanding of data types (as Gamma does);
- time-stamping of data;
- C libraries for the creation of custom clients;
- security access levels on data points; and,
- a confidence value for assigning fuzzy values to data points.

The Cogent DataHub is:

- a convenient way to disseminate real-time data;
- a RAM resident module holding current data;
- a proven solution with thousands of hours of installed performance; and,
- a great source of information for:
  - historical & relational database;
  - hard disk loggers; and,
  - Cascade Connect real-time connection to MS-Windows.

The Cogent DataHub is not:

- a historical database;
- a relational database;
- a hard disk logger;
- slow;
- a large memory requirement module; or,
- pre-configured.

Whenever multiple tasks are communicating there is a chance for a deadlock situation. The Cogent DataHub is at the center of many mission critical applications because it provides real- time data to its clients without the threat of being blocked on the receiving task. The Cogent DataHub never blocks on a task that is busy. The DataHub is always able to receive data from clients because it uses the **qserve** manager to handle outgoing messages. The DataHub only ever sends messages to the Cascade QueueServer program, which is optimized to never enter a state where it cannot accept a message from the Cogent DataHub.

13.3.4. Cogent DataHub Exceptions and Echos

When a new data point is sent to the Cogent DataHub the DataHub automatically updates its clients that are interested in the point. Some clients get information from the the DataHub on request only, by polling. Other clients register with the Cogent DataHub for changes in some or all points, called *exceptions*.

The Cogent DataHub not only allows its clients to register and receive exceptions on data points, but also provides a special message type called an *echo* that is extremely important in multi-node or multi-task applications.

When the Cogent DataHub receives a new data point it immediately informs its registered clients of the new data value. The clients will receive an asynchronous exception message. In some circumstances, the client that sent the new data value to the DataHub is also registered for an exception on that point. In this case, the originator of the data change will also receive an exception indicating the data change. When there are multiple clients reading and writing the same data point a client may wish to perform an action whenever another client changes the data. Thus, it must be able to differentiate between exceptions which it has originated itself, and ones which originate from other clients. The Cogent DataHub defines an echo as an exception being returned to the originator of the value change.

In certain circumstances, the lack of differentiation between exceptions and echos can introduce instability into both single and multi-client systems. For example, consider an application that communicates with another Lisp or MMI system, such as Wonderware's InTouch. InTouch communicates via DDE, which does not make the distinction between exceptions and echos. A data value delivered to InTouch will always be re-emitted to the sender, which will cause the application to re-emit the value to the Cogent DataHub. The DataHub will generate an exception back to the application, which will pass this to InTouch, which will re-emit the value to the application, which will send it to the DataHub, on so on. A single value change will cause an infinite communication loop. There are many other instances of this kind of behavior in asynchronous systems. By introducing echo capability into the DataHub, the cycle is broken immediately because the application can recognize that it should not re-emit a data change that it originated itself.

The echo facility is necessary for another reason. It is not sufficient to simply not emit the echo to the originating task. If two tasks read and write a single data point to the DataHub, then the DataHub and both tasks must still agree on the most recent value. When both tasks attempt to write the point, one gets

an exception and updates its current value to agree with the DataHub and the sender. If both tasks simultaneously emit different values, then the task whose message is processed first will get an exception from the first, and the first will get an exception from the second. In effect, the two tasks will swap values, and only one will agree with the DataHub. The echo message solves this dilemma by allowing the task whose message was processed second to receive its own echo, causing it to realize that it had overwritten the exception from the other task.

Appendix A. Function List

absolute_path	returns the absolute path of the given file.
access	checks a file for various permissions.
acos	finds the arc cosine of a number.
add_echo_function	assigns functions for echoes on a point.
add_exception_function	assigns functions for exceptions on a point.
add_hook	hooks a function to an event.
add_set_function	sets an expression to be evaluated when a given symbol changes value.
after	performs an action after a period of time.
alist_p	tests for association lists.
allocated_cells	gives the number of allocated and free cells.
and	is the same as the corresponding logical operator (&&).
append	concatenates several lists into a single new list.
apropos	finds all defined symbols in the current interpreter environment.
aref	returns an expression at a given index.
array	constructs an array.
array_p	tests for arrays.
array_to_list	converts an array to a list.
aset	sets an array element to a value at a given index.
asin	finds the arc sine of a number.
assoc	searches an association list for a sublist, using eq.
assoc_equal	searches an association list for a sublist, using equal.
at	performs an action at a given time, or regularly.
atan	finds the arc tangent of a number.
atan2	finds the arc tangent with two arguments.
atexit	evaluates code before exiting a program.
AutoLoad	allows for run-time symbol lookup.
autoload_undefined_symbol	checks undefined symbols for AutoLoad.
AutoMapFunction	maps a C function to a Gamma function.
autotrace_p	is for internal use only.
backquote	corresponds to a quote operator.
band	performs bitwise and operations.
basename	gives the base of a filename.
bdelete	deletes a single character from a buffer.
bin	converts numbers into binary form.
binsert	inserts a value into a buffer.
block_signal	starts signal blocking.
block_timers	blocks timer firing.
bnot	performs bitwise not operations.
bor	performs bitwise inclusive or operations.
breakpoint_p	is for internal use only.

`bsearch`	searches an array or list for a element.
`buffer`	constructs a buffer.
`buffer_p`	tests for buffers.
`buffer_to_string`	converts a buffer to a string.
`builtin_p`	is for internal use only.
`bxor`	perform bitwise exclusive or operations.
`caaar`	returns that element of a list.
`caadr`	returns that element of a list.
`caar`	returns that element of a list.
`cadar`	returns that element of a list.
`caddr`	returns that element of a list.
`cadr`	returns that element of a list.
`call`	calls a class method for a given instance.
`cancel`	removes a timer from the set of pending timers.
`car`	returns that element of a list.
`cd`	changes the working directory.
`cdaar`	returns that element of a list.
`cdadr`	returns that element of a list.
`cdar`	returns that element of a list.
`cddar`	returns that element of a list.
`cdddr`	returns that element of a list.
`cddr`	returns that element of a list.
`cdr`	returns that element of a list.
`ceil`	rounds a real number up to the next integer.
`cfand`	performs and operations with a confidence factor.
`cfor`	performs or operations with a confidence factor.
`char`	generates an ASCII character from a number.
`char_val`	generates a character's numeric value.
`chars_waiting`	checks for characters waiting to be read on a file.
`class_add_cvar`	adds new class variables.
`class_add_ivar`	adds an instance variable to a class.
`class_name`	gives the name of the class.
`class_of`	gives the class definition of a given instance.
`class_p`	tests for classes.
`ClearAutoLoad`	removes all `AutoLoad` rules.
`clock`	gets the OS time.
`close`	closes an open file.
`close_task`	closes a task opened by `locate_task`.
`conf`	queries confidence factors.
`cons`	constructs a cons cell.
`cons_p`	tests for cons cells.
`constant_p`	tests for constants.
`copy`	makes a copy of the top list level of a list.
`copy_tree`	copies the entire tree structure and elements of a list.

`cos`	returns the cosine of a number.
`create_state`	is part of the SCADALisp exception-driven state machine mechanism.
`date`	gets the OS date and time; translates seconds into dates.
`date_of`	is obsolete, see date
`dec`	converts numbers into base-10 form.
`defclass`	is the function equivalent of the statement: `class`.
`defmacro`	is a Lisp equivalent of the function: `macro`.
`defmacroe`	is a Lisp equivalent of the function: `macro`.
`defmethod`	is the function equivalent of the function: `method`.
`defun`	is a function equivalent of the statement: `function`.
`defune`	is a function equivalent of the statement: `function`.
`defvar`	defines a global variable with an initial value.
`delete`	removes an element from an array.
`destroy`	destroys a class instance.
`destroyed_p`	tests for destroyed instances.
`_destroy_task`	should never be used.
`dev_read`	is a modification of QNX 4 `dev_read`.
`dev_setup`	is obsolete, see `ser_setup`.
`difference`	constructs a list of the differences between two lists.
`directory`	returns the contents of a directory.
`dirname`	returns the directory path of a file.
`div`	divides two numbers, giving an integer result.
`dlclose`	closes an open dynamic library.
`dlerror`	reports errors in dl functions.
`dlfunc`	reserved for future use.
`DllLoad`	loads dynamic libraries.
`dlmethod`	reserved for future use.
`dlopen`	loads a dynamic library from a file.
`drain`	modifies end-of-file detection.
`enter_state`	is part of the SCADALisp exception-driven state machine mechanism.
`eq`	compares for identity and equivalence.
`equal`	compares for identity and equivalence.
`errno`	detects and numbers errors.
`error`	redirects the interpreter.
`eval`	evaluates an argument.
`eval_count`	counts evaluations made since a program started.
`eval_list`	evaluates each element of a list.
`eval_string`	evaluates a string.
`every`	performs an action every number of seconds.
`exec`	executes a program.
`exit_program`	terminates the interpreter.

`exit_state`	is part of the SCADALisp exception-driven state machine mechanism.
`exp`	calculates an exponent of the logarithmic base (e).
`fd_close`	closes a open file identified by a file descriptor.
`fd_data_function`	attaches a write-activated callback to a file.
`fd_eof_function`	attaches an `_eof_`-activated callback to a file.
`fd_open`	opens a file or device and assigns it a file descriptor.
`fd_read`	reads a buffer or string from an open file identified by a file descriptor.
`fd_to_file`	creates a file pointer from a descriptor.
`fd_write`	writes a buffer or string to an open file identified by a file descriptor.
`file_date`	gives the file modification date.
`file_p`	tests for files.
`file_size`	gives the file size.
`fileno`	creates a file descriptor from a pointer.
`find`	searches a list using the function: `eq`.
`find_equal`	searches a list using the function: `equal`.
`fixed_point_p`	tests for fixed-point reals.
`floor`	rounds a real number down to its integer value.
`flush`	flushes any pending output on a file or string.
`flush_events`	handles all pending events, then exits.
`fork`	duplicates a process.
`format`	generates a formatted string.
`free_cells`	returns the number of available memory cells.
`funcall`	provides compatibility with other Lisp dialects.
`function_args`	lists the arguments of a function.
`function_body`	gives the body of a user-defined function.
`function_calls`	tells how often a function was called during profiling.
`function_name`	gives the name of a function.
`function_p`	tests for functions.
`function_runtime`	gives the time a function has run during profiling.
`gc`	runs the garbage collector.
`gc_blocksize`	is for internal use only.
`gc_enable`	is for internal use only.
`gc_newblock`	is for internal use only.
`gc_trace`	controls the tracing of garbage collection.
`gensym`	generates a unique symbol.
`getcwd`	gets the current working directory.
`getenv`	retrieves the value of an environment variable.
`gethostname`	gets the computer’s host name.
`getnid`	returns the local node number.
`getpid`	returns the program ID.
`getprop`	returns a property value for a symbol.

`getsockopt`	gets a socket option.
`has_cvar`	queries for the existence of a class variable.
`has_ivar`	queries for the existence of an instance variable.
`hex`	converts numbers into hexadecimal form.
`init_async_ipc`	requests queue information from a task.
`init_ipc`	sets up necessary data structures for IPC.
`inp`	queries hardware ports (by byte).
`inpw`	queries hardware ports (by word).
`insert`	inserts a value at a given position.
`instance_p`	tests for instances.
`instance_vars`	finds all the instance variables of a class or instance.
`int`	converts numbers to integer form.
`int_p`	tests for integers.
`intersection`	constructs a list of all the elements found in both of two lists.
`ioctl`	performs control functions on a file descriptor.
`is_busy`	determines if a file is busy.
`is_class_member`	checks if an instance or class is a member of a class.
`is_dir`	determines if a file is a directory.
`is_file`	determines if a file exists.
`is_readable`	determines if a file is readable.
`is_writable`	determines if a file is writable.
`isend`	sends a synchronous message and doesn't wait for the result.
`ivar_type`	returns the type of a given instance variable.
`kill`	sends a signal to a process.
`length`	counts the number of elements in a list or array.
`list`	creates lists, evaluating the arguments.
`list_p`	tests for lists.
`list_to_array`	converts a list to an array.
`listq`	creates lists without evaluating the arguments.
`load`	loads files.
`load_lisp`	loads Lisp files.
`locate_task`	finds and connects to tasks by name.
`locate_task_id`	finds and connects to tasks by task ID and network node.
`lock_point`	locks or unlocks points.
`log`	calculates natural logarithms.
`log10`	calculates base 10 logarithms.
`logn`	calculates logarithms of a given base.
`long_p`	tests for long integers.
`macro`	helps generate custom functions.
`macro_p`	tests for macros.
`make_array`	creates an empty array.
`make_buffer`	creates a new, empty buffer.
`method_p`	tests for methods.
`mkdir`	creates a new sub-directory.

`mmap`	implements the C function call `mmap`.
`modules`	is obsolete, and returns nothing of value.
`name_attach`	attaches a name to a task.
`nanoclock`	gets the OS time, including nanoseconds.
`nanosleep`	pauses the interpreter for seconds and nanoseconds.
`nappend`	appends one or more lists, destructively modifying them.
`neg`	negates a number.
`new`	creates a new instance of a class.
`next_event`	blocks waiting for an event, and calls the event handling function.
`next_event_nb`	is the same as next_event, but doesn't block.
`nil_p`	tests for `nil` values.
`NoAutoLoad`	removes selected AutoLoad rules.
`not`	is the same as the corresponding logical operator (`!`).
`notrace`	turns tracing off.
`nremove`	removes list items, destructively altering the list.
`nreplace`	replaces elements in a list, using `eq`.
`nreplace_equal`	replaces elements in a list, using `equal`.
`nserve_query`	puts information from **nserve** into an array.
`nth_car`	iteratively applies the car functions to a list.
`nth_cdr`	iteratively applies the cdr functions to a list.
`number`	attempts to convert an expression to a number.
`number_p`	tests for numbers.
`oct`	converts numbers into octal form.
`open`	attempts to open a file.
`open_string`	allows a string to be used as a file.
`or`	is the same as the corresponding logical operator (||).
`outp`	writes values to hardware ports (by byte).
`outpw`	writes values to hardware ports (by word).
`parent_class`	returns the closest parent (base) of a class or instance.
`parse_string`	parses an input string.
`path_node`	gives the node number of a path in a QNX 2 path definition.
`pipe`	creates a pipe.
`point_locked`	indicates if a point is locked.
`point_nanoseconds`	gives the nanoseconds from point_seconds that a point value changed.
`point_seconds`	gives the time the point value changed.
`point_security`	gives the security level of a point.
`pow`	raises a base to the power of an exponent.
`pretty_princ`	writes to the standard output file, with formatting.
`pretty_print`	writes Lisp-readable output to the standard output file, with formatting.
`pretty_write`	writes an expression to a file, applying formatting.
`pretty_writec`	writes an expression to a file, applying formatting.

`princ`	writes to the standard output file.
`print`	writes Lisp-readable output to the standard output file.
`print_stack`	prints a Gamma stack.
`profile`	collects statistics on function usage and run time.
`prog1`	groups several statements into one expression.
`progn`	groups several statements into one expression.
`properties`	should never be used.
`pty`	runs programs in a pseudo-tty.
`ptytio`	runs programs in a pseudo-tty, using a termios structure argument.
`qnx_name_attach`	registers a local or global name.
`qnx_name_detach`	detaches a name.
`qnx_name_locate`	is an implementation of the C function `qnx_name_locate`.
`qnx_osinfo`	returns a class very similar to QNX 4 `struct_osinfo`.
`qnx_osstat`	lists processor loads and number of READY processes at each priority level.
`qnx_proxy_attach`	creates a proxy message for a process.
`qnx_proxy_detach`	removes a proxy.
`qnx_proxy_rem_attach`	creates a remote proxy message for a task.
`qnx_proxy_rem_detach`	removes a remote proxy.
`qnx_receive`	performs a QNX 4 `Receive`.
`qnx_reply`	replies to messages of type: `qnx_receive`.
`qnx_send`	implements QNX 4 `Send`.
`qnx_spawn_process`	is an implementation of the C function `qnx_spawn`.
`qnx_trigger`	tells a proxy to send its message.
`qnx_vc_attach`	establishes a virtual circuit between two processes on two computers.
`qnx_vc_detach`	detaches a virtual circuit.
`qnx_vc_name_attach`	attaches a virtual circuit with a name instead of a process ID number.
`quote`	corresponds to a quote operator.
`random`	generates random numbers from 0 to 1.
`raw_memory`	tells the amount of memory in use.
`read`	reads a Lisp expression from a file.
`read_char`	reads the next character from the input file.
`read_double`	reads the next double from the input file.
`read_eval_file`	reads a file, evaluating and counting expressions.
`read_existing_point`	retrieves points.
`read_float`	reads the next float from the input file.
`read_line`	reads a single line of text.
`read_long`	reads the next long value from the input file.
`read_n_chars`	reads and stores characters.
`read_point`	creates and/or retrieves points.
`read_short`	reads the next short value from the input file.

`read_until`	reads characters, constructing a string as it goes.
`real_p`	tests for reals.
`register_all_points`	registers an application to receive exceptions for all points.
`register_exception`	is not yet documented.
`register_existing_point`	registers an application to receive exceptions for a single existing point.
`register_point`	creates and/or registers an application to receive exceptions for a single point.
`registered_p`	tests for registered points.
`remove`	removes list items without altering the list.
`remove_echo_function`	removes an echo function from a symbol.
`remove_exception_function`	removes an exception function from a symbol.
`remove_hook`	removes a hooked function.
`remove_set_function`	removes a set function from a symbol.
`rename`	renames a file.
`require`	requires/loads files.
`require_lisp`	requires/loads Lisp files.
`required_file`	determines which files would be loaded.
`reverse`	reverses the order of list elements.
`root_path`	strips the final file or directory name from a path.
`round`	rounds a real number up or down to the nearest integer.
`rplaca`	replaces the car of a list.
`rplacd`	replaces the cdr of a list.
`run_hooks`	runs a hooked function.
`secure_point`	alters the security level on a point.
`seek`	sets the file position for reading or writing.
`send`	transmits expressions for evaluation.
`send_async`	transmits expressions asynchronously.
`send_string`	transmits strings for evaluation.
`send_string_async`	transmits a string asynchronously.
`ser_setup`	sets parameters for a serial port device.
`set`	assigns a value to a symbol, evaluating both arguments.
`set_autotrace`	is reserved for future use.
`set_breakpoint`	is reserved for future use.
`set_conf`	sets confidence factors.
`set_domain`	sets the default domain for future calls.
`set_random`	starts random at a different initial number.
`set_security`	changes the security level for the current process.
`setenv`	sets an environment variable for the current process.
`setprop`	sets a property value for a symbol.
`setprops`	lists the most recent property value settings.
`setq`	assigns a value to a symbol, evaluating the second argument.
`setqq`	assigns a value to a symbol, not evaluating any arguments.
`setsockopt`	sets a socket option.

`shell_match`	compares string text to a pattern.
`shm_open`	opens shared memory objects.
`shm_unlink`	removes shared memory objects.
`shorten_array`	reduces or expands the size of an array.
`shorten_buffer`	reduces the size of a buffer.
`signal`	defines an expression to be evaluated at an OS-generated signal.
`sin`	returns the sine of a number.
`sleep`	suspends execution for seconds.
`sort`	sorts a list or array, destructively modifying the order.
`sqr`	finds the square of a number.
`sqrt`	finds the square root of a number.
`stack`	lists all functions called so far.
`strchr`	searches a string for a character, returning the first location.
`strcmp`	compares strings, case-sensitive.
`strerror`	retrieves an error message.
`stricmp`	compares strings, case-insensitive.
`string`	constructs a string.
`string_file_buffer`	queries a string file for its internal buffer.
`string_p`	tests for strings.
`string_split`	breaks a string into individual words.
`string_to_buffer`	creates a buffer object from a string.
`stringc`	constructs a string in Lisp-readable form,
`strlen`	counts the number of characters in a string.
`strncmp`	compares two strings and return a numeric result, case-sensitive.
`strnicmp`	compares two strings and return a numeric result, case-insensitive.
`strrchr`	searches a string for a character, returning the last location.
`strrev`	reverses the order of characters in a string.
`strstr`	finds the location of a given substring.
`substr`	returns a substring for a given location.
`sym_alist_p`	tests for symbolic association lists.
`symbol`	constructs a symbol from a string.
`symbol_p`	tests for symbols.
`system`	treats its argument as a system command.
`tan`	returns the tangent of a number.
`taskdied`	calls a function when a task stops.
`task_info`	gets information from a task descriptor.
`taskstarted`	calls a function when a task starts.
`tell`	indicates file position.
`terpri`	prints a newline to an open file.
`time`	gives command execution times.
`timer_is_proxy`	controls timer handling in Gamma.

`tmpfile`	generates temporary output file names.
`tolower`	converts upper case letters to lower case.
`toupper`	converts lower case letters to upper case.
`trace`	turns tracing on.
`trap_error`	traps errors in the body code.
`true_p`	tests for truth value.
`unblock_signal`	ends signal blocking.
`unblock_timers`	unblocks timer firing.
`unbuffer_file`	causes a file to be treated as unbuffered on both input and output.
`undefined_p`	tests for undefined values.
`undefined_symbol_p`	tests for undefined symbols.
`union`	constructs a list containing all the elements of two lists.
`unlink`	deletes a file.
`unread_char`	attempts to replace a character to a file for subsequent reading.
`unregister_point`	stops echo and exception message sending.
`unwind_protect`	ensures code will be evaluated, despite errors in the body code.
`usleep`	suspends execution for microseconds.
`wait`	waits for process exit status.
`when_echo_fns`	indicates the functions for echos on a point.
`when_exception_fns`	indicates the functions for exceptions on a point.
`when_set_fns`	returns all functions set for a symbol.
`whence`	gives input information.
`write`	writes an expression to a file.
`write_existing_point`	writes values to existing points.
`write_n_chars`	writes characters from a buffer to a file.
`write_point`	writes point values, creating points if necessary.
`writec`	writes a Lisp expression to a file.

Appendix B. GNU Lesser General Public License

GNU Lesser General Public License

Version 2.1, February 1999

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method:

1. we copyright the library, and
2. we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author’s reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the *Lesser* General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

Section 0

This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

Section 1

You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Section 2

You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. The modified work must itself be a software library.

b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

   (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

Section 3

You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

Section 4

You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

Section 5

A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

Section 6

As an exception to the Sections above, you may also combine or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work

during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable “work that uses the Library”, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user’s computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the “work that uses the Library” must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

Section 7

You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

Section 8

You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who

have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Section 9

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

Section 10

Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

Section 11

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

Section 12

If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

Section 13

The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

Section 14

If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY Section 15

BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

Section 16

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the library’s name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library ’Frob’ (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990 Ty Coon, President of Vice

That’s all there is to it!

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.