Golden Common LISP

GOLD HILL COMPUTERS, INC.

GOLDEN COMMON LISP

Version 1.01

Gold Hill Computers
163 Harvard Street
Cambridge, Massachusetts 02139

Gold Hill Computers provides this publication "as is", without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Gold Hill Computers may make improvements or changes in this publication, or in the product and programs described in this publication, at any time and without notice.

LISP is copyrighted by Addison-Wesley Publishing Company, Inc. COMMON LISP Reference Manual is copyrighted by Digital Equipment Corporation.

'GOLDEN COMMON LISP' is a registered trademark of Gold Hill Computers. 'GCLISP', 'GMACS', and 'GOLDEN EMACS' are trademarks of Gold Hill Computers. 'San Marco LISP Explorer' and 'LISP Explorer' are trademarks of San Marco Associates. 'ZETALISP' is a trademark of Symbolics, Incorporated. 'IBM', 'IBM PC', and 'PC-DOS' are registered trademarks of International Business Machines Corporation. 'PC XT' and 'PC AT' are trademarks of International Business Machines Corporation. 'COMPAQ' is a trademark of COMPAQ Computer Corporation. 'MS-DOS' is a registered trademark of Microsoft, Incorporated. 'Smalltalk' is a trademark of Xerox Corporation. 'Mouse Systems' is a trademark of Mouse Systems Corporation. 'Intel 8088' and 'Intel 8087' are trademarks of Intel Corporation.

Printed in the United States of America.

ISBN 0-917589-05-X

GOLDEN COMMON LISP
Version 1.01

Upgrade Instructions

The accompanying items are the materials for upgrading your GCLISP Version 1.00 to Version 1.01.

These items are included:

1. Five (5) diskettes, each labeled "GCLISP Version 1.01" 1 (Master, 2 Utilities-1, 3 Utilities-2, 4 San Marco LISP Explorer Viewer, and 5 San Marco LISP Explorer Slides); and

2. A packet of pages for insertion to your GCLISP user documentation binder.

To upgrade your user documentation:

1. *Replace* the following items in the D-ring binder of Version 1.00 documentation with the corresponding items from the documentation update packet:

   - Title/copyright page, half-title page, and Contents page

   - Installation Guide (entire document to be replaced by Installation Guide Version 1.01 and "Golden Common LISP: Version 1.01 Installation Guidelines")

   - Tutorial Guide (entire document)

   - Title/Preface page and pp. 1 - 2 of the Users' Guide

   - Title pages of Reference Manual and Appendices

   - "A Quick Start-Up of GOLDEN COMMON LISP"

2. *Remove* the "Distribution Notice" (if it is present) from the back of the binder.

3. *Add* these items at the back of the binder:

   - Release Note GCL0100 - 1

- Release Note GCL0101 - 1

(Note that Release Note GCL0100 - 1 is a reduced-size reprint of the original previously sent to you, for easy insertion into the binder.)

4. *Correct* the Users' Guide, page 50: read **"Ctrl-X Ctrl-X"** for **"Ctrl-Z Ctrl-X"**.

To install your GCLISP Version 1.01 software: Follow the instructions in the Installation Guide, Version 1.01. and "Golden Common LISP: Version 1.01 Installation Guidelines".

March 15, 1985

A QUICK START-UP OF GOLDEN COMMON LISP

If this is the first time you have used GOLDEN COMMON LISP and you are eager to try the software, this short guide will show you how to start up the system and run the LISP Explorer and GMACS editor before turning to the full users' documentation.

After your initial exploration, please refer to the GCLISP Installation Guide for instructions about how to install and configure your system. Also, send in your Registration Card so that we can automatically notify you about new software releases.

GOLDEN COMMON LISP requires an IBM PC, PC XT, PC AT, or 100%-PC-compatible computer with at least:

- one double-density/double-sided diskette drive;
- 512K bytes of memory;
- A PC-DOS (or MS-DOS) operating system, Version 2.0 or higher (including Version 3.0).

The GCLISP Program License Agreement envelope contains five write-protected diskettes licensed for use on a single machine. The following directions assume a minimal PC configuration, with a single diskette drive and a monochrome display.

To explore GCLISP, follow the steps below. What you type to the computer appears in bold-face (e.g., **gclisp**). To enter a keychord like **Alt-E**, press and hold the **Alt** key, then hit the **E** key. If at any time you have a question, turn to the Installation Guide.

First, start the DOS operating system. If you have problems doing this, turn now to your IBM PC DOS Manual (or the equivalent for your computer).

Introduction to the San Marco LISP Explorer

Insert Master	Insert the GCLISP Master diskette in drive A:. Type **A:** to set the current drive to A:.
Start GCLISP	Type **gclisp** to load the GCLISP interpreter. This takes roughly half a minute. Type **R** when asked whether you want to install, un-install, or run GCLISP. The GCLISP Top-Level prompt (*) will appear shortly.
Explore	Type **Alt-E** to load the LISP Explorer. (Loading takes about two minutes. The system will prompt you to swap diskettes.) The LISP Explorer takes you on a self-guided tour of the world of LISP. To exit the LISP Explorer, type function key **F1**.
Exit GCLISP	Type **(exit)** to leave the GCLISP environment and go back to DOS. (Ignore the error message for now, as you have not yet configured your system.)

Introduction to the GMACS Editor

Insert Master	(See above.)
Start GCLISP	(See above.)
Enter GMACS	Type **Ctrl-E** to enter the GMACS editor. (This takes about one minute. The system will prompt you to swap diskettes.)
Get GMACS help	Type **Alt-H** to see the various types of GMACS Help available. Type **A** followed by **file** to find out all the editor commands for files.
Learn GMACS	Type **Alt-H T** to load a file that teaches you about GMACS.
Exit GMACS	Type **F1** to exit GMACS and return to GCLISP.
Exit GCLISP	Type **(exit)** to leave the GCLISP environment and go back to DOS.

At any time while in the GCLISP interpreter, you can type **Alt-H** to see the Top-Level Help screen.

When you are done exploring, please see the Installation Guide for important information about GCLISP.

GOLDEN COMMON LISP

Version 1.01

GOLDEN COMMON LISP

CONTENTS

PREFACE

GOLDEN COMMON LISP, or GCLISP, is a COMMON LISP training and programming environment for personal computers, designed to accommodate both new and experienced LISP programmers.

The GOLDEN COMMON LISP package comprises software tools and publications to train LISP novices and to support the development of advanced COMMON LISP application programs:

- The GCLISP interpreter implements a major subset of COMMON LISP functionality, observing most COMMON LISP standards and conventions.

- The San Marco LISP Explorer, an on-line interactive tutorial by San Marco Associates, teaches LISP programming and Artificial-Intelligence techniques.

- The EMACS-style editor GMACS is a full-screen, LISP-intelligent text editor for program development. It is complemented by high-level program debugging utilities.

- On-line help is available for every GCLISP function and variable.

- The book <u>LISP</u>, by Patrick H. Winston and Berthold Klaus Paul Horn (Second Edition; Addison-Wesley, 1984), is the most widely-used text on LISP.

- The <u>COMMON LISP Reference Manual</u>, by Guy L. Steele Jr. (Digital Press, 1984), is the definitive COMMON LISP language specification.

In addition, this binder of user documentation includes both tutorial materials and reference materials for GCLISP users. The documents included here are:

<u>GCLISP Installation Guide</u>

This is the document to read first. It contains an inventory of GOLDEN COMMON LISP components and operating requirements. Instructions on how to use the GOLDEN COMMON

LISP diskettes and a guide to the documentation are also included.

GCLISP Tutorial Guide

This document provides instructions for using the San Marco LISP Explorer. The LISP Explorer is geared to the beginning programmer, drawing on concepts developed in Winston and Horn's LISP.

GCLISP Users' Guide

This users' guide explains how to use the features of the GOLDEN COMMON LISP environment: the interpreter, the GMACS editor, the on-line help facilities, and the debugging utilities. The guide also provides commentary on basic and often-used LISP structures and functions. It explains principles and ideas of LISP, and provides instructions for creating and testing LISP programs. A sample application illustrates the design and construction of a GOLDEN COMMON LISP program.

GCLISP Reference Manual

This manual defines the syntax and semantics of the GOLDEN COMMON LISP language. It has been designed to complement the COMMON LISP Reference Manual, using the same table of contents, format, and notational conventions.

GCLISP Appendices

Appendix A, "Error Messages", lists the error messages produced by the GOLDEN COMMON LISP interpreter.

Appendix B, "Glossary", provides a glossary of LISP terminology and other technical terms used in the documentation.

Appendix C, "The Window System", documents the interface to the GOLDEN COMMON LISP window system.

Appendix D, "Compatibility Notes", documents points of divergence between GOLDEN COMMON LISP and the COMMON LISP standard.

ACKNOWLEDGMENTS

GOLDEN COMMON LISP has come about largely through the efforts of Harold Ancell, Gerald R. Barber, Judith A. Bolger, Martin J. Broekhuysen, Hilary C. Chan, Cody F. Curtis, Stanley P. Curtis, Nick Gall, Carl Hewitt, John Kam, Joseph D. Pehoushek, Dominique M. Schroeder, John A. Seamster, John A. Teeter, Eugene Wang, and Chaka.

The package would not have been completed without the expertise of Patrick H. Winston, Daniel C. Brotsky, and Karen A. Prendergast, who developed the San Marco LISP Explorer and gave us valuable input in the design of GOLDEN COMMON LISP. Ms. Prendergast also provided the painting for the cover design of the GOLDEN COMMON LISP package.

The following individuals and groups deserve special acknowledgment for their contributions:

Guy L. Steele Jr., who wrote the COMMON LISP language specification, and allowed us to use the book's original name of the COMMON LISP Reference Manual.

John Osborn and Chase Duffy of Digital Press, who worked closely with us to produce a version of the COMMON LISP Reference Manual for our package.

David K. Wessel and Ellen D. Rawlings of Addison-Wesley Publishing Company, who helped us to include the book LISP (Second Edition), by Patrick H. Winston and Berthold Klaus Paul Horn.

Daphne Fogg of CSA Press, who worked hard to help us deliver a quality product under a demanding and changing delivery schedule.

Daniel J. Dawson, who designed the graphics for the GOLDEN COMMON LISP package, and remarkably made it all come together.

GOLDEN COMMON LISP
INSTALLATION GUIDE

Version 1.01

Table of Contents

Installation Guide

1 Introduction

GCLISP is designed to be easy to install on a variety of PC configurations.

You should follow the instructions in each section of this Guide (see the Guide's Table of Contents) in order to ensure that your GCLISP package is complete, and is properly installed and configured.

If you have any problems understanding the terms or conventions used in this Guide, you should turn to section 10, "Some Terminology".

If you still have problems, you should turn to section 9, "What to Do if Things Go Wrong".

2 Minimum System Requirements

Make sure that your PC is capable of running GCLISP.

This is the minimum configuration required to run GCLISP:

- An IBM PC, PC XT, PC AT, or 100%-PC-compatible computer;
- 512K bytes of memory;
- A PC-DOS (or MS-DOS) operating system, Version 2.0 or higher (including Version 3.0);
- A 5-1/4" double-sided/double-density diskette drive and diskette drive controller; and
- Either a Monochrome Display Adapter and a Monochrome Display, or a Color/Graphics Monitor Adapter and a Color/Graphics Monitor.

The following configuration options are also supported by GCLISP:

- A second 5-1/4" double-sided/double-density diskette drive and diskette drive controller;
- A hard disk and disk drive controller;
- A Mouse Systems PC Mouse; and
- An Intel 8087 Numeric Processor Extension.

Note: All disk drives, diskette drives, drive controllers, display/monitors, and display/monitor adapters listed above must be IBM or IBM-compatible. Operation of the copy-protected GCLISP diskettes depends upon the full IBM-compatibility of the drives and drive controllers.

3 An Inventory of the GCLISP Package

Check that your GCLISP package contains these items:

- LISP, by Patrick H. Winston and Berthold Klaus Paul Horn (Second Edition; Addison-Wesley Publishing Company, 1984).

- The COMMON LISP Reference Manual, by Guy L. Steele Jr. (Digital Press, 1984).

- A D-ring binder containing the following:

    * Installation Guide (this document)

    * Tutorial Guide

    * Users' Guide

    * Reference Manual

    * Appendices

    * Customer Protection Plan

    * Two four-pocket diskette sleeves

    * Program License Agreement envelope (containing five write-protected diskettes)

    * A Quick Start-Up of GOLDEN COMMON LISP

    * Release Notes GCL0100 - 1 and GCL0101 - 1

The five diskettes contained in the Program License Agreement envelope are as follows:

- GCLISP Master diskette

- GCLISP Utilities 1 diskette

- GCLISP Utilities 2 diskette

- San Marco LISP Explorer Viewer diskette

- San Marco LISP Explorer Slides diskette

4 Starting the DOS Operating System

Start your DOS operating system.

If you do not know how to start your DOS operating system, follow the instructions in your IBM PC DOS manual (or its equivalent for your machine).

You do not need to restart DOS every time you want to run GCLISP. We suggest that you *do* start it afresh before installing GCLISP, to ensure that no other program will affect the installation process.

5 Installing GCLISP

5.1 Installation and Copy-Protection

GCLISP is copy-protected. The copy-protection mechanism enables you to install GCLISP on diskettes or on a hard disk, while preventing unauthorized duplication of the software.

The number of authorized installations is pre-set on the original distribution diskettes. An authorized installation can be made on either a hard disk or on diskettes, at your choice.

GCLISP can be run either from the original distribution diskettes or from an authorized installation. However, you should always install GCLISP, and then run it from the installed copy, keeping the original product diskettes safely stored away. In case of accidental damage to the installed copy, the originals are then available for running GCLISP. Also, the original Master diskette is required whenever you want to un-install an installed copy.

In the remainder of this Guide, several terms are used for convenience. A product diskette is one of the five original (distributed) diskettes you purchased. A working diskette is a diskette you have produced by installing on it the contents of a product diskette, using the installation procedure described in section 5.3 below. During the installation procedure, the diskette where GCLISP is to be installed is also called a target diskette. The diskette drive where the target diskette is inserted is called the target drive or the installation drive.

Since the installation procedure consists essentially of copying the product diskettes, a working diskette or a hard-disk installation is also called a working copy.

The following installation instructions guide you through the normal installation procedure. See Appendix A to this Guide, "Un-Installation and Error Messages", for instructions on how to un-install an installed GCLISP, and for explanation of error messages that may be displayed during the procedures.

5.2 Installing GCLISP on a Hard Disk

Before installing GCLISP on your hard disk, make sure that:

1. There are at least 1,800,000 bytes free on your hard disk (this is how much room the GCLISP system occupies). You can determine the number of free bytes on drive C: (for example) by using the DOS command **chkdsk**, as follows:

   C>chkdsk c:

   If there is insufficient room, you will have to delete some existing files from the hard disk.

2. There is no directory named **\gclisp** on the hard disk containing files which you want to save. By default, GCLISP will be installed in **\gclisp** (during the installation procedure, you may, if you want, name the logical drive and the directory where GCLISP will be installed). If there is already such a directory, *all the files in it will be deleted* before the new files are installed.

To install GCLISP on your hard disk:

1. Insert the GCLISP product Master diskette in drive A:, and make drive A: the current drive.

2. Enter the command **gclisp** at the DOS prompt:

   A>gclisp

3. Type **I** when asked whether you want to install, un-install, or run GCLISP.

4. Follow the other prompts displayed on the ensuing display screens.

The installation process takes about 10 minutes to transfer the GCLISP system from the five diskettes to the hard disk. Any time during the installation process, you may abort the installation by typing **Ctrl-Break** and then typing 'Yes'. (**Ctrl-C** instead of **Ctrl-Break** if your keyboard lacks a Break key.)

After the installation process has successfully completed, it will set the current directory to the GCLISP default drive and directory for the hard-disk installation, and then start GCLISP. (Future starts can take place from that directory directly, as described in section 6.1 below.)

5.3 Installing GCLISP on Diskettes

To install GCLISP on diskettes:

1. Prepare five working diskettes by formatting them, using the DOS command **format**.

2. Insert the GCLISP product Master diskette in drive A:, and make drive A: the current drive.

3. Enter the command **gclisp** at the DOS prompt:

   A>gclisp

4. Type **I** when asked whether you want to install, un-install, or run GCLISP.

5. Follow the other prompts displayed on the ensuing display screens.

The installation process takes about 15 minutes to transfer the GCLISP system from the five product diskettes to the formatted working diskettes. Any time during the installation process, you may abort the installation by typing **Ctrl-Break** and then typing 'Yes'. (**Ctrl-C** instead of **Ctrl-Break** if your keyboard lacks a Break key.)

After the installation process has successfully transferred the GCLISP system to the formatted diskettes, it will start GCLISP from the new working copy on these diskettes. (Future starts should take place from the working copy directly, as described in section 6.2 below.)

6 Starting GCLISP

Since the installation process starts GCLISP automatically from the new working copy, you can ignore this section the first time around. But in general, you should follow one of the procedures below to start GCLISP.

There are two cases: starting GCLISP from a hard-disk installation, or starting GCLISP from a diskette installation.

6.1 Starting GCLISP from a Hard-Disk Installation

1. Make drive C: the current drive by entering the following command at the DOS prompt (for example, the prompt **A>**):

   A>c:

2. Make the **gclisp** directory the default directory by entering the following command:

   C>cd \gclisp

3. Enter the GCLISP environment by entering the following command:

   C>gclisp

6.2 Starting GCLISP from a Diskette Installation

1. Insert a working copy of the Master diskette in drive A:.

2. Make drive A: the current drive by entering the following command in response to the DOS prompt (for example, the prompt **B>**):

   B>a:

3. Make \ (the root) the default directory by entering the following command:

   **A>cd **

4. Enter the GCLISP environment by entering the following command:

   A>gclisp

7 Configuring GCLISP

The very first time that GCLISP is started from a working copy, the display will appear as follows:

```
GOLDEN COMMON LISP, Version 1.01
Copyright (C) 1984, 1985 by Gold Hill Computers

; Reading file INIT.LSP

Initialization file loaded.
This GCLISP has not been configured,
    type (CONFIGURE-GCLISP).
Type Alt-H for help
Top-Level
* _
```

Note: The message "This GCLISP has not been configured, type (CONFIGURE-GCLISP)" will not appear once you have configured your system using **configure-gclisp**.

To configure GCLISP for use on your system, type the following at the GCLISP prompt (the * character):

```
* (configure-gclisp)
```

(GCLISP begins processing the command as soon as the right parenthesis is typed; you do not need to hit the Enter key.)

configure-gclisp will inform GCLISP about your system by asking you questions concerning the type of monitor on your system and the amount of memory to reserve for DOS. Each question is accompanied by a full explanation. You may go over the questions several times until you are completely satisfied with your answers. When you exit, your GCLISP will be configured. (The amount of memory you have specified to be reserved for DOS will not take effect until the next invocation of GCLISP.)

You can run **configure-gclisp** as often as needed to reflect changes in your system's resources and their allocation.

8 Where to Go from Here

Congratulations on successfully installing GCLISP on your system!

8.1 Becoming a GCLISP Registered User

Now before you get too caught up exploring the world of GCLISP, you should send in the self-addressed GCLISP Registration Card (located in the Customer Protection Plan at the back of this binder). This card establishes you as a registered user, which entitles you to receive written notification of upgrades to GCLISP, replacements for missing or damaged parts, and four free newsletters.

Please fill out this card and return it to us now.

Note: The "software serial number" to be entered on the Registration Card is found on a white label near the top left of your diskettes. The ISBN number at the top right is not the software serial number.

8.2 Guide to the Documentation

In general, the documentation is designed to be read sequentially in the order of its appearance in the binder.

If you are new to LISP or if you would like to brush up on LISP arcana, you should go to the Tutorial Guide (next in this binder), where you will be introduced to the San Marco LISP Explorer. The LISP Explorer, in conjunction with the book LISP, will provide you with an excellent introduction to the fundamentals of LISP programming.

If you are an experienced LISP programmer, you may want to bypass the Tutorial and proceed directly to the Users' Guide to get a feel for the environment provided by GCLISP.

Once you have read the Users' Guide and are ready to program, you will want to read the GCLISP Reference Manual together with the COMMON LISP Reference Manual to familiarize yourself with the capabilities of GCLISP in particular and COMMON LISP in general. Note that most of the material in the Reference Manual is available on-line via the GCLISP help facilities.

If you have a particular problem or area for investigation, use the following heuristics for finding the information you want:

- Look through the Table of Contents of each document in this binder to locate where a topic is written about;
- Consult the Index of each document, for references to pages where significant words or phrases appear;
- Look through Appendix B, "Glossary", for the meanings of technical terms;
- Type **Alt-H** to access the on-line help facilities;
- Type **Alt-E** to use the San Marco LISP Explorer.

9 What to Do if Things Go Wrong

Don't panic.

Review this Installation Guide and make sure you have followed the installation, startup, and configuration procedures correctly.

If you are having trouble with installation, see section A.2, "Errors and Error Messages", in this Guide's Appendix A, "Un-Installation and Error Messages".

If the problem appears to be with your computer system, or with the distribution diskettes, or you can't get GCLISP started or configured, try to find your problem in the Troubleshooting Guide below, and take the specified remedial action.

If you have started GCLISP, but are encountering problems using it, consult the Release Notes included in this binder. Also consult Appendix A, "Error Messages", in the Appendices at the back of this binder.

If you still can't solve your problem, call or write for Customer Technical Support:

Gold Hill Computers
Customer Technical Support
163 Harvard Street
Cambridge, MA 02139

Phone: (617) 492-2071

Troubleshooting: A Short Guide

PROBLEM	REMEDIAL ACTION
A package component is missing or damaged.	Fill out the Replacement Order Card (located in the Customer Protection Plan at the back of this binder) and send it to our Customer Technical Support address (above).

Files are damaged or missing on
an original GCLISP diskette.

Take the remedial action for damaged components, above.

You aren't sure that your system meets
the minimum requirements for running GCLISP.

Attempt the installation process. If your system doesn't meet the minimum requirements, you should receive either the message **Program too big to fit in memory** (see below), or a message described in your IBM PC DOS manual (or its equivalent for your computer).

While starting GCLISP, you receive the message
Program too big to fit in memory

You must have at least 480K bytes of *available* memory in order to run GCLISP. You can determine the amount of available memory on your system using the DOS **chkdsk** command. The available memory may be limited by the presence of device drivers or a RAM drive, for instance.

You don't know how to start DOS
or how to enter DOS commands.

This installation guide assumes that you are familiar with the basic use of the DOS operating system on your PC. If you are not, you should consult your IBM PC DOS manual before continuing with the installation process.

GCLISP starts, but prints out an error message
instead of the GCLISP prompt, *.

For a detailed explanation of the error and the appropriate remedial action, consult Appendix A, "Error Messages", in the Appendices at the back of this binder.

10 Some Terminology

10.1 Some Definitions

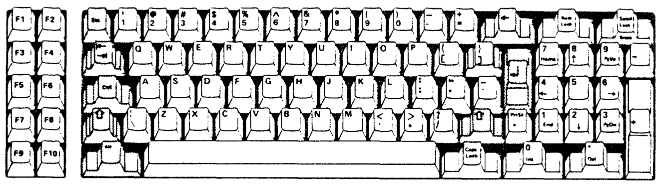
The following table defines certain terms that appear frequently in this Guide. If the term that you are looking for is not defined here, see Appendix B, "Glossary", at the back of this binder.

TERM	MEANING
cursor	The cursor is a blinking mark (usually an underline, '_') on the display that indicates where the next typed character will appear. The cursor is usually to the right of the last character typed.
enter	For a DOS command, the phrase "enter the command **foo**" means typing the characters **f**, **o**, **o**, and then hitting the Enter key (see the next subsection). For a GCLISP command, the phrase "enter the command **(foo)**" means typing the characters **(**, **f**, **o**, **o**, **)**, *without* hitting the Enter key.
initialization	Initialization is the process of loading an operating system or software package into a computer in order to run it.
prompt	A prompt is a character (or characters) that appears on the left-hand side of the display when a system is waiting for a user command. There are different prompts for different systems. For example, **A>** is the default DOS prompt, while ***** is the normal GCLISP prompt.

10.2 The IBM Keyboard

This section introduces the IBM PC keyboard. It defines the names of certain keys and key groups and explains how they are used within GCLISP. GCLISP makes special use of the keyboard, so you should at least skim this section even if you are already quite familiar with the keyboard.

Here is a diagram of the IBM PC keyboard.



IBM PERSONAL COMPUTER KEYBOARD

The keyboard is divided into three areas of keys: a "Typewriter Area", a "Numeric/Cursor-Control Keypad Area", and a "Function Key Area".

Typewriter Area

These keys occupy the large middle area of the keyboard. Most of the keys resemble the keys of an ordinary typewriter, and function like typewriter keys.

However, the following keys perform special actions in GCLISP:

Enter key — This key is located on the right-hand side of the Typewriter Area in the same location that the Return key occupies on a typewriter. It is marked with a bent, left-pointing arrow to suggest the action of a Return key.

At the DOS prompt, the Enter key is typed at the end of a command in order to tell DOS to begin processing the command. In other words, the Enter key "enters" a command (hence the name). When typing text in the GMACS editor, the Enter key acts like the Return key on a typewriter: it moves the cursor to the first character position of the next line.

Rubout key — This key appears on the upper right-hand side of the Typewriter Area, just above the Enter key. It is marked with a long left-pointing arrow. (It is easy to confuse this key with the cursor-control key that is marked with a short left arrow.) This key is also known as the Backspace key.

Any time you are entering text, the Rubout key can be used to delete the characters to the left of the cursor.

Control key — This key is located in the left middle of the Typewriter Area, above the Shift key. It is marked **Ctrl**. The Control key works like the Shift key: You press and hold down the Control key, and then type another key.

Throughout the GCLISP documentation, the prefix **Ctrl-** is used with a key that is to be typed with the Control key. For example, **Ctrl-F** refers to pressing and holding down the Control key, and then typing the **F** key.

Alternate key — This key is located in the lower left of the Typewriter Area, below the Shift key. It is marked **Alt**. The Alternate key works like the Shift key: You press and hold down the Alternate key, and then type another key.

Throughout the GCLISP documentation, the prefix **Alt-** is used with a key that is to be typed with the Alternate key. For example, **Alt-F** refers to pressing and holding down the Alternate key, and then typing the **F** key.

Parentheses keys — The open and close parentheses are the shift positions of the **9** and **0** keys, top row right in the Typewriter Area. In GCLISP, parentheses surround all commands. Note that when the closing parenthesis of a command is typed, GCLISP immediately begins processing the command (you do not need to hit the Enter key).

Escape key — This key is located on the upper left-hand side of the Typewriter Area, just above the Tab key. It is marked **Esc**. At the GCLISP prompt, hitting **Esc** will delete the current input. In the GMACS editor, it is used in place of the Enter key in certain situations.

Print-Screen key — This key is located to the right of the right-hand Shift key. It is marked **PrtSc**. When this key is struck with the Shift key held down, DOS prints the information on the display to the printer. If the information contains any graphics, the printer must be compatible with the IBM Graphics Printer. Note that **Shift-PrtSc** toggles copying to the printer in DOS, but not in GCLISP.

Numeric/Cursor-Control Keypad

These keys are located on the right-hand side of the keyboard:

Numeric Lock key
This key is located at the top of the keypad. It is marked **Num Lock**. It acts as a toggle, switching the keypad between use as a Numeric keypad and a Cursor Control keypad.

Scroll-Lock - Break key
This key is located at the top-right of the keypad. It is marked **Scroll Lock** on top and **Break** on the front. Holding down the **Ctrl** key and hitting this key will cause GCLISP to "break" the currently executing function (see the Users' Guide for more information).

Cursor-Control keys
These keys consist of the four arrow keys: The Page Up key (labeled **Pg Up**), the Page Down key (labeled **Pg Dn**), the **Home** key, and the **End** key. In the GMACS editor, these are used to move the cursor around on the display. In the San Marco LISP Explorer, they are used to move through the lessons.

Function Keys

These keys are located on the left-hand side of the keyboard. They are labeled **F1** to **F10**.

At the DOS prompt, they are used for simple editing of the command line. At the GCLISP prompt, they merely generate graphics characters. In the GMACS editor and the San Marco LISP Explorer, they are used as command keys.

Appendix A

Un-Installation and Error Messages

The installation process described in section 5 runs interactively, prompting you to type disk drive and directory identifiers, and to insert diskettes, as needed. The prompts are mostly self-explanatory. However, note that the informational output line "Diskettes MUST NOT have a write protect tab" refers to all of the working diskettes and to the product Master diskette. The write-protect tabs should be left on the other product diskettes.

A.1 Un-Installation

GCLISP can be un-installed -- that is, removed -- from a hard disk or a diskette where it has been installed. Un-installation of a working copy makes it possible to re-install a new working copy (to the same medium or elsewhere). This useful feature helps to protect you against the consequences of diskette wear, and also enables you to switch an installation from one medium to another.

To perform un-installation, insert the *product Master diskette* (the *original, distributed* diskette, not a working copy) in diskette drive A:, set the default drive to A:, and start GCLISP. Your display screen will shortly ask whether you want to install GCLISP, or un-install GCLISP, or simply run GCLISP. If you choose to un-install a hard-disk installation, you will also be able to choose whether to delete from the hard disk all of the GCLISP files, or only the principal program files.

Important note: Performing a DOS RESTORE operation on the root directory of a hard disk on which GCLISP has been installed can damage the GCLISP copy-protection system. Therefore, you should un-install GCLISP before restoring to the root directory.

A.2 Errors and Error Messages

Errors may occur while you are installing or un-installing. If an error message is displayed, find it in the following list and take the specified remedial action. If the installation or un-installation procedure has aborted, it can then be re-started. (In some instances it will continue after your correction, for example after removal of a write-protect tab from a diskette.)

If a displayed error message is not found in the list below, contact Gold Hill Computers.

Note that these messages are related specifically to installation and un-installation. Other possible problems with your computer system, or with diskettes, or with configuring or starting GCLISP, were described in section 9 above, "What to Do if Things Go Wrong".

Diskette is Write Protected

> The write-protect tab has been left on the target diskette. Remove the target diskette from the drive, take off the write-protect tab, and put the diskette back in the drive.

Remove write-protect tab from diskette

> (Same as the preceding message, for the target diskette or the product Master diskette.)

Not Enough Space

> There is too little space on the target diskette (or on the hard disk) to create the working copy. Use only a freshly formatted diskette for the target diskette in a diskette installation. For a hard disk installation, this message means that some files must be deleted from the hard disk to make room for GCLISP.

Not enough space on target disk

> (Same as the preceding message. This message may also appear for an invalid drive specification.)

Drive Not Ready

There is no diskette in the target diskette drive. Insert a (formatted) diskette in the drive, and press the Enter (or Return) key to continue with the installation process.

Not enough storage to run the Install program

Installation or un-installation requires at least 96K available memory in your machine.

Invalid drive specification

You have specified a non-existent drive. Verify that the physical and logical drive assignments are correct, and specify only drives which are on your system.

Product is already installed. Install aborted.

During the installation process, GCLISP was found already installed on the target diskette or disk drive. There is no need to install to this medium.

Product never installed. UNinstall aborted.

You've tried to un-install GCLISP from a diskette or a hard disk where it is not currently installed.

Product protection system damaged

The copy-protection mechanism is damaged. Contact Gold Hill Computers for a replacement.

Install Terminated Error Code = *nnnn*

Contact Gold Hill Computers to remedy a situation resulting in this error message with a 4-digit error code. Note: The code 6010 may appear if a write-protect tab is left on.

Unauthorized Duplicate
Load Failed *nn*
or
Load Failed Error Code = *nnnn*
or
Unauthorized Duplicate (Code *nnnn*)

Contact Gold Hill Computers.

Index

GOLDEN COMMON LISP

VERSION 1.01 INSTALLATION GUIDELINES

This short document supplements the Golden Common LISP Installation Guide, Version 1.01, found in the binder of user documentation included in every purchase of GCLISP Version 1.01.

For more details about installation, including the exact instructions for running the installation procedure, please refer to the Guide. The procedure has been designed and programmed so that making each installation should be a routine process.

(A) Before sitting down to do any installation, please observe the following guidelines and cautions.

1. An installation can be made to diskette or to hard disk. You choose which during the installation procedure.

2. Any installed copy can be un-installed, making that copy available for installation to another hard disk or another set of diskettes.

3. GCLISP Version 1.01 is fully installable to the IBM and COMPAQ families of personal computers including the IBM PC, PC XT, PC AT, and Portable (but not the PCjr); and the COMPAQ, COMPAQ Plus, and COMPAQ DeskPro. It is also fully installable on 100%-compatible computers including the AT&T PC 6300, Columbia PC, Olivetti, some Zenith and Corona PC's, and Tandy 1000's and 1200's.

   On other computers, including those manufactured by Sperry, Leading Edge, ITT, Televideo, Panasonic, and Eagle, it is not fully installable. It is also not fully installable to certain hard disks, including the Datamac 33Mb, Great Lakes, Iomega Bernoulli Box, Tecmar, Cameron 10Mb, Sunol 25Mb, and Alloy. If your personal computer or your hard disk is one of these, call us for technical information first.

4. IBM has published the fact of a possible incompatibility between the IBM PC AT and the rest of the IBM PC family, including the PC and the PC XT. Double-density diskettes which are written on in a quad-density diskette drive on

the PC AT may thereafter not be readable in the double-density diskette drives of PC's and PC XT's. Since every installation of GCLISP -- either a hard-disk or a diskette installation -- involves writing to the GCLISP distribution Master diskette, a distribution Master diskette used to install GCLISP from the quad-density drive of an AT may thereafter be unreadable on any PC or PC XT.

This is a vendor-hardware problem which could create a problem for GCLISP installations. Our tests of GCLISP installations on the PC AT have not encountered it. However, to minimize the risk, do this: *perform all installations on PC AT's after any other installations.*

5. Before, during, and after installation and un-installation runs, handle the distribution Master diskette with care. It is needed for every installation and un-installation run.

(B) As you prepare to install on hard disk, be aware that 480K bytes of RAM memory must be available on the target machine for GCLISP to run. Run the DOS command **chkdsk**. The last line of output displayed from **chkdsk** shows the available RAM memory ("bytes free").

(C) During the installation procedure, observe the following:

1. You can abort the installation procedure at any time by typing **Ctrl-Break** (or **Ctrl-C** if your machine lacks a **Break** key). If you do this before the GCLISP interpreter has been installed, the process can be re-started from the beginning. If you abort the process after the GCLISP interpreter has been installed, then you should first un-install and then re-install. The interpreter has been installed if, and only if, the file GCLISP.COM is present in the target installation directory you have chosen (usually the directory **C:\GCLISP** on hard disk, or the root directory on a diskette).

2. These are the most common causes of problems during the installation procedure:

   - *Less than 480K bytes of RAM memory is available in your machine.*

     A machine with 512K bytes or more may have less than 480K available because other programs are resident in memory when GCLISP is started. Use **chkdsk** as described in **(B)** above to find out if too little memory is available. A RAM disk, a spooler, a terminal emulator, device drivers, or a popular program such as Borland International's Sidekick program may be occupying memory. Remove the offending program and re-start the installation procedure.

- *A write-protect tab is on the distribution Master diskette or on any installation target diskette.*

  These diskettes are written on during the installation procedure. Remove the offending tab and re-start the installation procedure from the beginning.

- *The diskette drive heads on the source diskette drive are unclean or un-aligned.*

  Rarely, but sometimes, this inhibits installing. Make sure that the drive heads are clean and well-aligned.

- *The DOS command processor (the program COMMAND.COM) is not found during a hard-disk installation.*

  COMMAND.COM is needed by the installation procedure. The symptom that it is not available is either (i) empty target directories (LISPLIB, EXAMPLE, etc.) after an installation that has run without any sign of trouble; or (ii) the message "Cannot find file CR.CR" during the installation. To verify directly that COMMAND.COM is not available, start GCLISP from the distribution Master diskette; choose the Run option ("R"); and, when the prompt * appears, type **Ctrl-D** to invoke DOS. The message "Failed: COMMAND.COM not found ..." will appear if COMMAND.COM is not available.

  The common cause of this problem is booting your computer from a DOS system diskette, without having a copy of COMMAND.COM on the hard disk. To remedy the problem:

  > Be sure that a copy of COMMAND.COM is in the root directory on the hard disk (if necessary putting it there by copying it from a DOS system boot diskette).
  >
  > Be sure that the environment variable COMSPEC is set to access this hard-disk copy

of COMMAND.COM, by inserting in your machine's CONFIG.SYS file the command:

```
shell=c:\command.com c:\ /P
```

(For further explanation, see the DOS technical reference manual for your machine.)

When you encounter a problem without a quick solution, consult the Installation Guide, including its Appendix A, "Uninstallation and Error Messages".

(D) Post-installation cautions:

- After a hard-disk installation, observe the caution in Appendix A regarding RESTORE operations on the hard disk.
- If any installed GCLISP diskettes show signs of wear after a period of time, un-install that copy and re-install it to new diskettes.

GOLDEN COMMON LISP
TUTORIAL GUIDE

Version 1.01

The San Marco LISP Explorer

The GOLDEN COMMON LISP Tutorial consists of the San Marco LISP Explorer[1], an interactive, self-contained exploration of the basic programming concepts and strategies of LISP.

The LISP Explorer is organized like a slide show: each topic is presented as a sequence of screens, much like a tray of slides. You choose trays and slides using a screen menu and keys on your PC keyboard.

To invoke the LISP Explorer from within the GCLISP environment, type the GCLISP command **(explore)** -- including the parentheses -- or the keychord **Alt-E**. This places you in the LISP Explorer environment. The function keys **F1** - **F4** and **F10** can be used to orient yourself and to move around in the environment:

F1 — **"Return to GCLISP"**
This ends the LISP Explorer session and returns you to the GCLISP environment.

F2 or **Alt-H** — **"The Key Diagram"**
This summarizes how to get around in the LISP Explorer environment using the cursor motion keys (**Right Arrow, Left Arrow, Up Arrow, Down Arrow, PgUp, PgDn, Home** and **End**) and these five function keys.

F3 — **"Itinerary World"**
This displays the topics of the LISP Explorer in the form of a menu. Use the cursor motion keys **Right Arrow, Left Arrow, Up Arrow,** and **Down Arrow** to locate the tray you want to invoke, and then **F3** to invoke it.

F4 — **"Primitive World"**
This displays a list of LISP primitives and enables rapid access to a tray in which each is introduced. Use the cursor motion keys to locate the primitive you want information about, and then **F4** to access a tray where that

----

1. "San Marco LISP Explorer" and "LISP Explorer" are trademarks of San Marco Associates.

primitive is discussed.

F10 **"Practice World"**
This enables you to practice what you have learned by typing input to the GCLISP interpreter from within the LISP Explorer environment.

The message "Writing usage history" appears briefly on the screen when you exit from the LISP Explorer. This usage-history file, USAGE.LSP, enables the LISP Explorer to keep track of the last-viewed slide and the set of trays which you have already accessed. Any time you re-enter the LISP Explorer, you will be presented with the slide and tray you were viewing when you last exited. Any time you view the itinerary, using **F3**, the itinerary menu will mark the trays you have already completed.

If there is not enough space to load the LISP Explorer when you try to enter it, you will receive an informational message and the LISP Explorer will not be started. This will happen if you have used up a great deal of the GCLISP workspace, for example by loading the GMACS editor. When this occurs, you can end the current GCLISP session by typing **(exit)**, start a new GCLISP session by typing **gclisp**, and then type **Alt-E** to enter the LISP Explorer.

The LISP Explorer includes trays of slides on these topics:

Preview
Using the Controls
The Itinerary
Abstraction
From Bowls to Lists
Atoms and Lists
LISP Evaluates Forms
Lists Can Be Forms
Symbol Can Be Forms
Quoting Stops Evaluation
Access Procedures
Selector Procedures
Combining List Selectors
The Simplest Constructor
Making Simple Procedures
Watching Procedures Work
More List Constructors
Still More List Constructors
Making More Procedures
Exploiting Analogies
Testing with Predicates
The Equality Predicate
The Data Type Predicates
The List Predicates
The Numeric Predicates

Simple Branching
General Branching
Combining Predicates
Repeating by Recursing
Recursing Twice
The Individual Inspector
The Group Inspector
Binding Variables
Evaluating Sequences
Following Paths to Files
Editing Files
Reading Files
Repeating by Iterating
Repeating by Mapping
Procedural Arguments
Nameless Procedures
Using Association Lists
Using Properties
Using Arrays
Using Structures
Simple Printing
Simple Reading
Formatted Printing
Boxes and Arrows
Using Backquote
Translating with Macros
Optional Arguments
Approaching New Worlds
The Blocks World
Search
Pattern Matching
Rule-based Experts
Natural Language
Intelligent Data Bases
Moving On

The San Marco LISP Explorer is self-guiding. With this short introduction, you can invoke it for LISP instruction any time you are in the GCLISP environment.

GOLDEN COMMON LISP
USERS' GUIDE

Version 1.01

PREFACE

This Users' Guide introduces the GCLISP environment. It teaches you how to type and evaluate GCLISP functions in the interpreter, and how to use the GMACS editor for constructing LISP programs. It also explains the use of the on-line help facilities and the debugging utilities. Finally, it includes the development of a sample application that introduces various aspects of GCLISP programming.

If you are completely new to LISP, you may want to use the San Marco LISP Explorer (see the Tutorial Guide) to introduce yourself to LISP concepts before putting them to work in the GCLISP environment.

Table of Contents

Chapter 1

The GCLISP Interpreter

LISP stands for *List Processing*. Lists are the principal means for organizing both data and program structures in LISP. Because both programs and data are lists, program structures can be treated as data: that is, as input to other programs. Consequently, LISP functions can analyze other LISP functions, and can even build new LISP functions.

Another aspect of LISP's flexibility is the extent to which the user is able to define new LISP data and modify existing ones. As Bernard Greenberg has said about LISP:

> LISP objects are often used to model real-world objects. Like real-world objects, LISP objects have properties and relations to each other. A typical real-world object, like a house, has a color, a number of stories, the street it is on, the people who live in it, and other qualities and quantities as "properties" ... In a LISP program, we might have one object represent each house we were dealing with .... LISP allows us to define, establish, utilize and change the various properties and relations of groups of objects.[1]

This chapter introduces you to interaction with the GCLISP *interpreter*. The interpreter is the main program of GCLISP. It establishes and maintains your GCLISP environment. This is the environment within which you type in LISP forms, or load files of LISP forms, for evaluation. From the interpreter environment, you can call on the GCLISP tutorial for instruction, or invoke the GMACS editor to create program files. The program debugging utilities run in the interpreter environment, and so does the on-line help system. During much of your work in GCLISP, you are in direct communication with the interpreter.

----

1. Bernard Greenberg, "Notes on the Programming Language LISP" (Student Information Processing Board, Massachusetts Institute of Technology; 1976)

1.1 Entering GCLISP

To enter the GCLISP environment at any time, first set your DOS working directory. If you have installed GCLISP on a hard disk, set the working directory to C:\GCLISP. If not, the working directory should be set to logical disk drive A:, where you have inserted the installed working copy of the GCLISP Master diskette.

Then enter the command **gclisp** in response to the DOS command prompt:

```
C>gclisp<ENTER>
```

The display here shows the operating system prompt and your **gclisp** command. (The prompt shows the logical disk drive, assumed here to be the drive **C:**.) **<ENTER>** stands for typing the key labeled with a bent arrow (sometimes also called RETURN, or CARRIAGE RETURN or CR, or ENTER). A display screen like the following will result:

```
GOLDEN COMMON LISP, Version 1.01
Copyright (C) 1984, 1985 by Gold Hill Computers

; Reading file INIT.LSP

Initialization file loaded.
This GCLISP has not been configured,
   type (CONFIGURE-GCLISP).

Type Alt-H for Help
Top-Level
* _
```

The title and copyright lines, and two lines about initialization, are informational output from GCLISP.

The message "This GCLISP has not been configured, type (CONFIGURE-GCLISP)" appears only if you have not yet run the configuration program (see the Installation Guide). You should run this first, before continuing in GCLISP. (Then the message will not appear again.)

The one-line guide to invoking on-line help about GCLISP and the "Top-Level" line inform you that the GCLISP interpreter has been invoked. The final line is the initial prompt to you from GCLISP (*), and the cursor mark (_) showing where your input will be typed. At this point, you are in the GCLISP environment. You can enter LISP forms for evaluation; or request on-line help about the environment by typing the keychord **Alt-H**.[2] You can also invoke the San Marco LISP Explorer or the GMACS editor.

Occasionally while you are working in GCLISP, the lower-left corner of the display screen will flash the letters "GC" for a few seconds. This indicates that GCLISP is performing "garbage collection" on the workspace: reclaiming storage in the workspace so that it is available for the allocation of new LISP objects. This is an automatic process which will not affect your interaction, except to slow the interpreter's response to your typing while the indicator is flashing.

Throughout the Users' Guide and other user documentation, we will illustrate your interaction with GCLISP with "sample screens" like the one above. These will be examples of actual input-output dialogues. A sample screen will always be marked by left and bottom borders, as just shown. User input will always be shown in lower-case letters. Output from GCLISP may be in upper-case or lower-case (or mixed).

With rare exceptions, you should be able to reproduce these dialogues exactly from within your GCLISP environment.

----

2. The notation **Alt-H** means "the **H** key is pressed while the **Alt** key is held down." See section 2.1.7 regarding this and other *keychords*.

1.2 Exiting from GCLISP

When you want to exit from the GCLISP environment (immediately, or after doing any amount of work) type in **(exit)**. This returns you to the operating system:

```
* (exit)
C>
```

Note the parentheses in the input to the interpreter above. The closing parenthesis signals the end of input to the interpreter, and invokes immediate evaluation of the input. <ENTER> need not be typed.

(exit) resets the entire GCLISP environment. You should use **(exit)** only when you are done working in GCLISP for a while, or when you need more computer memory for non-GCLISP applications. To execute a temporary exit, preserving the GCLISP environment, use the GCLISP function **dos**, described below.

1.2.1 Exit and Re-Entry

The function **exit** ends the current GCLISP session, returning you to the command processor in the operating-system environment. You can then enter DOS commands in this environment again; and you can at any time re-start GCLISP with the **gclisp** command to the operating system.

However, during any GCLISP session, you may occasionally want to execute a DOS command. It would waste time to end the GCLISP session, run the DOS command, and re-start GCLISP. You can more easily run the DOS command from within GCLISP without[3] terminating the current session.

To do this, use the GCLISP function **dos**, as in this example:

----

3. If your system does not have a hard disk, the diskette containing the DOS command processor -- the file COMMAND.COM -- should be in the current drive when you invoke DOS from within GCLISP.

```
* (dos "copy foo.lsp bar.lsp")

NIL
* _
```

That is: at the GCLISP prompt, enter the DOS command line, for example **copy foo.lsp bar.lsp**, as an argument to the function **dos**. The DOS command line is enclosed in double quotes. GCLISP sends the command out to DOS for execution. No matter what the command is, the return value of the GCLISP function **dos** is **nil**, provided there are no errors in the DOS command line. (Otherwise the return value is a numerical error code from DOS. See sections 1.5 - 1.6 regarding return values of evaluated functions.) When DOS has executed the command, the return value is printed to your screen, and then the interpreter is ready as usual for your next GCLISP input form. (Any output from the DOS command line will be printed to the screen and will be displayed temporarily before the return value is printed.)

More generally, you can execute two or more DOS commands in sequence and still return to the current GCLISP environment:

```
* (dos)

C>copy foo.lsp bar.lsp<ENTER>
          1 File(s) copied

C>time<ENTER>
Current time is 19:23:14.21
Enter new time:<ENTER>

C>exit<ENTER>

NIL
* _
```

That is: the function call **(dos)**, with no arguments, places you in the DOS environment for as long as you like, without ending the current GCLISP session. When you are done working in DOS, the DOS command **exit** restores the GCLISP environment as it was when you left. (The display will not look exactly as just shown, because GCLISP also resumes printing to the screen exactly where it left off.)

The keychord **Ctrl-D** has the same effect as the function call **(dos)**.

Note: In the "temporary DOS environment" provided by the command **(dos)**, use the **exit** command to return to GCLISP. Don't give the **gclisp** command. This would establish a new GCLISP session without ending the suspended one.

1.3 On-Line Help

You can get on-line help at any time when typing input.

To see the on-line help guide, type the keychord **Alt-H** (the Alt key held down while the H key is pressed). The help guide appears, showing the types of help available and the two principal GCLISP applications:

```
To invoke one of the following GCLISP applications,
type the indicated keychord:

  Alt-E    The LISP Explorer, an on-line tutorial
 Ctrl-E    The GMACS Editor

To get help in one of the following areas,
type the indicated keychord:

  Alt-K    "Keys" - Displays a list of the actions
           invoked by special keys and keychords.

  Alt-A    "Apropos"  - Lists all symbols whose names
           contain a specified string. Prompts for the string.

  Alt-D    "Documentation" - Displays the on-line documentation
           for a specified function, variable, or type name.
           Prompts for the name.

  Alt-L    "Lambda-List" - Displays the arguments for a
           specified function.  Prompts for the function name.
_______________________________________________________________
* _
```

For more information about on-line help, see Chapter 3, "On-Line Help Facilities", and also section 1.4 below. (The GMACS editor has its own, separate on-line help facility; see Chapter 2.)

1.4 Keychord Commands to the Interpreter

Certain commands to the interpreter are invoked by special keyboard keys and keychords. The complete list of keychord commands is displayed when you type **Alt-K**, for "Keys" help. It appears as follows:

```
These are the GCLISP keychord commands:

   Alt-A            Apropos a string
   Alt-D            Document a function, variable, or type
   Alt-E            Enter the LISP Explorer
   Alt-H            Invoke On-line Help
   Alt-K            Display this list of keychord commands
   Alt-L            Display the lambda-list of a function
  Ctrl-B            Backtrace the execution stack
  Ctrl-C            Unwind to Top-Level
  Ctrl-D            Invoke the DOS command processor
  Ctrl-E            Invoke the GMACS editor
  Ctrl-G            Go up one error level
  Ctrl-L            Clear the display screen
  Ctrl-P            Continue from a break
  Ctrl-Break        Enter into a break level
  Ctrl-NumLock      Halt typeout to screen (any key continues)
  Rubout            Delete the preceding character
  Esc               Delete the current input line

These keychords can be typed at any time.

* _
```

The specialized help commands (**Alt-K**, **Alt-A**, **Alt-D**, and **Alt-L**) and the application commands (**Ctrl-E** and **Alt-E**) are described in more detail when you type **Alt-H**. The individual keys **Rubout** and **Esc** are convenience aids for typing input. Other keychord commands give information about the environment, or alter the environment. These are explained elsewhere in this chapter and the rest of the Users' Guide.

1.5 The "Read-Eval-Print" Loop

LISP program structures are processed by a LISP evaluator, which consists of a function called **eval**. The user interacts with the evaluator primarily through a loop that includes two other functions besides **eval**: **read** and **print**. Not surprisingly, this loop is referred to as the *read-eval-print loop*. The program that implements this loop is known as the *listener*.

The loop consists of three steps, in order:

(1) Read
(2) Evaluate
(3) Print

In detail, these steps operate as follows:

(1) Read. The function **read** transforms the characters typed at the keyboard into LISP objects. For example, if the sequence of characters "f", "o", "o" is typed, **read** returns the symbol named FOO.[4] If the sequence "(", "+", " ", "2", " ", "3", ")" is typed, **read** returns a list containing the symbol **+** and the integers **2** and **3**.

(2) Evaluate. The object returned by **read** is passed to the function **eval**, which *evaluates* (or *interprets*) the object and returns the result(s) of the evaluation. For example, when **read** passes the list **(+ 2 3)** to **eval**, **eval** returns the integer **5** as the result.

(3) Print. The results of the evaluation are passed to **print**, which outputs the printed representation of the results to the screen. For example, if eval passes the integer **5** to **print**, **print** outputs the character **5** to the screen.

----

4. The *LISP reader*, or just the *reader*, is the program which implements the **read** function. It changes lower-case letters to upper-case letters, except when reading character-string data. Consequently, you can type a symbol in either lower-case or upper-case letters, or any mix of cases, without affecting the interpretation.

1.6 Evaluation of LISP Forms

The examples in this section illustrate the basic form of user interaction with the listener as described above.

These simple examples will be familiar to a LISP programmer. If you are completely new to LISP, you may need to call on the GCLISP Tutorial, or Winston and Horn's book LISP, for more extended introductions to the language.

The simplest form you can enter to the listener is a number:

```
* 2<ENTER>

2
* _
```

The number **2** is read and evaluated; the result, **2**, is printed to the screen. A number always evaluates to itself. (In LISP a form which evaluates to itself is called a *self-evaluating form*.)

The last line of the output is the prompt, signalling that the loop has been completed and the listener is ready to receive your next input.

Function evaluation is illustrated by simple addition. The addition function is a compiled function in GCLISP,[5] represented by the symbol **+**. To add two numbers, we can type in as shown:

```
* (+ 2 2)
4
* _
```

----

5. For a complete specification of all of the functions and variables supported by GCLISP, see the GCLISP Reference Manual.

The + function is evaluated with the arguments **2** and **2**, and the result, **4**, is printed to the screen. As with all LISP functions, the function name precedes the function arguments; and the resulting function call is entered as a list: that is, enclosed in parentheses.

Note that in the current example, the closing parenthesis in the input signaled the completion of an input form. No **<ENTER>** input was needed. The input reader recognized and assembled the form, and passed it to the evaluator. In the preceding example, however, the input reader needed the terminating **<ENTER>** to recognize the end of the input form (any white space following the input would also have signaled the reader). In each case, the **print** function prints its result to the screen on the next new line.

A number evaluates to itself; a function call evaluates to the result of applying the function to the values of its arguments, as just illustrated. A symbol, however, is interpreted as representing a variable; and it cannot be evaluated unless it has previously been assigned (or *bound to*) a value. Unless a variable is bound to a value, its evaluation causes an error. The following screen illustrates an evaluation error with the unbound symbol **two**.

```
* two<ENTER>


ERROR:
Unbound variable: TWO
1> _
```

(Note the different prompt **1>**, representing a new level of the listener. Listener levels are described in section 1.8 below; and errors and error messages are described in section 1.9.)

To assign the symbol **two** a value, use the **setf**[6] function:

```
|
|
| * (setf two 2)
| 2
| * _
|
|____________________________________
```

Now the symbol **two** can be evaluated:

```
|
|
| * two<ENTER>
|
| 2
| * _
|
|____________________________________
```

We can now perform the addition function using the symbol **two** rather than the number **2**:

```
|
|
| * (+ two two)
| 4
| * _
|
|____________________________________
```

A symbol can be bound to a new value at any time with the **setf** function. Suppose we change the value of the symbol **two** to the numeric value **3**:

```
|
|
| *(setf two 3)
| 3
| * _
|
|____________________________________
```

----

6. We use **setf** rather than **setq** because **setf** is more general than **setq**, and for this reason, more in accord with the philosophy of COMMON LISP.

Now if we add **two** and **two**, the result is the number **6**:

```
* (+ two two)
6
* _
```

For a final example, consider defining a new function named "plus". In this illustration, "plus" will be a limited version of the GCLISP function **+**. That is, it will be defined as a function of two arguments which adds its arguments and returns the result, as **+** does. (**+** is somewhat more powerful than "plus", because **+** can be applied to more than two arguments, and it also performs type checking on its input arguments.)

To define "plus", use the GCLISP function **defun**:

```
* (defun plus (a b) (+ a b))
PLUS
* (plus 2 2)
4
* (plus two two)
6
* _
```

Here, the result of evaluating the first form was the function name **plus** (output in upper-case). Then we input a function call: the function **plus** applied to the arguments **2** and **2**. This evaluated, as expected, to **4**. However, **plus** applied to the symbol **two** (for both arguments) evaluated to **6**, since the most recent value bound to **two** was **3**.

A major part of LISP programming is developing LISP forms which you expect to use again and again. Any such form can be defined as a function using **defun**. Thereafter, to use the function, you have only to enter the function name together with specific arguments.

1.7 System Variables for Tracking Listener Actions

The listener maintains several variables which provide a useful history of its most recent actions. These variables have short, easily-remembered names composed from the characters "*", "/", and "+". At any time during a GCLISP session, you can use any one of these variables.

One of these is the variable *, which always has the value[7] returned from the last evaluated form. The following sample screen illustrates its use:

```
* (min  (max 5 10 25) (max 7 49))
25
* *<ENTER>

25
* (setf answer *)
25
* answer<ENTER>

25
* _
```

The first line in the sample screen computes the maximum of the numbers 5, 10, and 25 (which is 25); computes the maximum of the numbers 7 and 49 (which is 49); and then computes the minimum of these two results (25). Then the variable * evaluates to 25. The **setf** line sets the value of the symbol **answer** to the current value of *, or 25. Then the variable **answer** evaluates to 25.

* represents only one value returned from an evaluated function. If the function returns more than one value, * represents just the first return value. To retrieve (in the form of a list) all of the values returned from a multi-valued function, use / instead of *. For example, the **truncate**

----

7. Note that the symbol * also represents the multiplication function in CGLISP. (And is also displayed as a prompt.) Be careful not to confuse these meanings from the start.

function divides its second argument into its first argument; and returns the quotient as the first value and the remainder as the second value:

```
* (truncate 17 4)
4
1
* *<ENTER>

4
* (truncate 17 4)
4
1
* /<ENTER>

(4 1)
* _
```

That is: the function call **(truncate 17 4)** returns the values **4** and **1** (quotient and remainder); and ***** then returns **4** (the first return value). But / directly following the function call returns the *list* with the two return values as its elements.

The value of the variable + is the most recently read LISP form, as shown in this example:

```
* (min (max 5 10 25) (max 7 49))
25
* +<ENTER>

(MIN (MAX 5 10 25) (MAX 7 49))
* (min  (max 5 10 25) (max 7 49))
25
* (setf problem +)
(MIN (MAX 5 10 25) (MAX 7 49))
* problem<ENTER>

(MIN (MAX 5 10 25) (MAX 7 49))
* _
```

Note carefully: * and / take their values from the most recent error-free *evaluation;* but + takes its value from the most recent error-free *reading*. That is, + is updated every time an error-free input form is read, whether the form can be evaluated without error or not. However, only a form that can be evaluated without error will change the value of * or /.

The variables **, //, and ++ have the corresponding meanings for the *next-to-last* evaluated form (or the next-to-last read form). And the variables ***, ///, and +++ have the corresponding meanings for the *third-from-last* evaluated (or read) form. The following table summarizes these variables.

VARIABLE	MEANING
*	Represents the first value returned from the last evaluated LISP form.
**	Represents the first value returned from the next-to-last evaluated LISP form.
***	Represents the first value returned from the third-from-last evaluated LISP form.
/ // ///	Represents a list of all the returned values from the last evaluated form, or the next-to-last, or the third-from-last.
+ ++ +++	Represents the last-read form, or the next-to-last, or the third-from-last.

1.8 Listener Levels

When you enter GCLISP via the command **gclisp** from the operating system, you are placed at "level 0" of the listener, or "Top-Level". This level can be recognized by the prompt * appearing on your display screen.

During your interaction with the listener -- typing in of forms, evaluation, and printing of results to the screen -- "deeper levels" (or "lower levels") of the listener may be invoked. These are numbered 1, 2, ... (higher numbers for deeper levels). You can recognize these by the numbered prompts **1>**, **2>**, ...

Only one level of the listener is active at any time; and you interact only with that level. The GCLISP user interface behaves the same at every level: accepting forms, evaluating them, and printing the results.

How is a deeper level activated? There are two possible ways. The first is by an error in user input. This example appeared in section 1.6, when an unbound symbol was entered:

```
* two<ENTER>


ERROR:
Unbound variable: TWO
1> _
```

There is no reason to stay at level 1 in this case. You input the keychord **Ctrl-G** or the function call **(clean-up-error)** to return to level 0:

```
1> <Ctrl-G>
Back to: Top-Level
* _
```

Level 1 disappears, and you are returned to where you left off at level 0, as shown by the prompt *****. A subsequent error at level 0 would invoke a new level 1.

Similarly, an error in input at level 1 invokes a level 2 listener. A return from there via **clean-up-error** or **Ctrl-G** returns to the level 1 where it was suspended (from which you may return to level 0 again). And similarly for deeper levels.

An error is an unintended way to invoke a deeper level. The second way to invoke a deeper level is deliberate. Inputting the function call **(break)** or the keychord **Ctrl-Break** invokes the next deeper level:

```
* (break)

BREAK, (CONTINUE) to continue

1> _
```

This is useful as a program debugging technique (see Chapter 4, "Debugging in GCLISP"). Internal data about the suspended level is accessible to you at the deeper level, and may be useful in detecting and fixing program bugs.

Just as when the deeper level was invoked by error, you can continue processing as you like at the deeper level and return to the higher level when you choose. In this case, however, the return is not via **Ctrl-G** but via **continue** or **Ctrl-P**:

```
1> <Ctrl-P>
NIL
* _
```

Note carefully the difference between an *error* invocation of a deeper level and a *break* invocation. The returns are different:

> **(clean-up-error)** or **Ctrl-G** returns from an error
> **(continue)** or **Ctrl-P** returns from a break

Ctrl-C is a useful, more powerful return from a deeper level entered either by error or deliberately. It returns to level 0 immediately, discarding any and all intervening deeper levels.

1.9 Common User Errors and GCLISP Error Messages

Both new and experienced LISP programmers make frequent errors when inputting LISP forms to the listener. GCLISP responds immediately to user errors. The usual response to an error is an error message printed to the screen, and an invocation of the next deeper level of the listener.

This section describes the most common errors and the responses to them. A complete listing of error messages is in Appendix A, "Error Messages".

Unbound variable. This interaction was described in section 1.8:

```
* two<ENTER>

ERROR:
Unbound variable: TWO
1> <Ctrl-G>
Back to: Top-Level
* _
```

In this instance, the symbol **two** did not have an assigned value.

Undefined function. Just as a variable must be bound to a value before it can be evaluated, a function name must be defined before it can be used in a function call.

The error message **Undefined function** results when you attempt to use in a function call a name which hasn't been defined as a function. This error is often caused by mistaking a variable name for a function name. Suppose, for instance, that **foo** was assigned a value, but not defined as a function; and then you attempt to use **foo** as a function name in a function call:

```
* (setf foo 2)
2
* (foo)

ERROR:
Undefined function: FOO
While evaluating: (FOO)
1> foo<ENTER>

2
1> <Ctrl-G>
Back to: Top-Level
* _
```

Remember that the parentheses around **foo** indicate to the LISP listener that **(foo)** is a function call; while **foo** (no parentheses) is interpreted by the listener as a variable.

<u>Wrong number of arguments in a function call.</u> If we define a function **foo** to take two arguments, and apply it to three arguments, we receive the message: **Too many arguments for: FOO**, as in this example:

```
* (defun foo (a b) (+ a b))
FOO
* (foo 6 1 4)

ERROR:
Too many arguments for: FOO
While evaluating: (FOO 6 1 4)
1> <Ctrl-G>
Back to: Top-Level
* _
```

Wrong type of argument. You receive the error message **Wrong type argument** if you use one type of LISP object as an argument to a function that expects a different type of object as an argument. This occurs, for instance, if you use a number for an argument when the function expects a symbol. The function **get**, for example, takes two arguments: a symbol and an object of any type. If we input a number rather than a symbol for the first argument:

```
* (get 2 'size)

ERROR:
GET: wrong type argument: 2
A SYMBOL was expected.
1> <Ctrl-G>
Back to: Top-Level
* _
```

The following table summarizes the error messages just described.

MESSAGE	EXPLANATION
Unbound variable: *foo*	The symbol *foo* was used as a variable, but had no value assigned to it.
Undefined function: *foo*	The symbol *foo* was used as a function name in a function call, but had not been defined as a function name.
Too many arguments for: *foo*	*foo* was defined as a function name; but in a function call to *foo*, too many arguments were supplied.
foo: wrong type argument: *X*	The type of an argument *X* supplied in a call of the *foo* function does not match the type of argument required by the function definition of *foo*.

1.10 Loading Input Files

A LISP program consists of a sequence of LISP forms, written one after the other.

For a program of any size, it makes no sense to type in the forms one at a time from your console, in the style shown so far in this Guide. A program of even a few lines will more likely be typed first into an on-line file; and then the entire file is input to GCLISP for reading and evaluation. This is the conventional way of writing and debugging LISP programs.

Doing this requires two main tools. One is an on-line editor for creating and modifying the on-line program file. The GCLISP on-line editor is called GMACS; and the next chapter in this Guide is a detailed guide to using GMACS. The other tool is the LISP function **load**, which directs GCLISP to read and evaluate the contents of a program file. **load** is described in this section.

Suppose that (using GMACS) a program file called **FOO.LSP** has been created, with these contents:

```
(+ 2 3)
(defun bar (a b)
  (* a b))
(bar 4 5)
```

That is: **FOO.LSP** consists of three LISP forms. The first form is a simple addition; the second defines the function **bar** as simple multiplication; the third is a function call to **bar** with the arguments **4** and **5**.

To have the file **FOO.LSP** read and its forms evaluated, give a **load** function call at your console. The result looks like this:

```
* (load "foo.lsp")
; Reading file C:\GCLISP\FOO.LSP


#.(PATHNAME "C:\\GCLISP\\FOO.LSP")
* _
```

That is: **load** takes the name of the program file as an argument. The name must be delimited by quote characters ("). The **load** call prints the "Reading file" informational message; and, when reading (and evaluation) has been successfully completed, the full pathname of the file is printed to the screen.

Several language conventions shown in this sample screen will be unfamiliar to the LISP novice. For a short explanation of their meanings, see section 1.11, "Table of COMMON LISP Language Conventions". Note in particular the double backslash, \\. This signifies that the reader has expanded a pathname built with single backslashes. Since the backslash character is a language convention which specifies that the following character is to be taken literally, two successive backslashes are needed to represent a backslash to the listener. (For an explanation of pathnames, see the tray entitled "Following Paths to Files" in the San Marco LISP Explorer.)

Unlike a **read-eval-print** loop, the **load** function does not automatically print to the screen the results of evaluating the forms in the input file. Thus, though the forms in **FOO.LSP** were evaluated, the screen did not show the results. To print the returned values on the screen, include the **:print** option in the **load** function call:

```
* (load "foo.lsp" :print t)
; Reading file C:\GCLISP\FOO.LSP

5
BAR
20
#.(PATHNAME "C:\\GCLISP\\FOO.LSP")
* _
```

Compare this screen with the contents of **FOO.LSP** to verify the evaluations.

The **:print** option helps you to locate errors in the program file. Suppose, for example, that in the function definition of **bar** in **FOO.LSP**, the last parenthesis were missing, so that it would look like this:

```
(defun bar (a b)
  (* a b)
```

Now load this "defective" version of **FOO.LSP**, using **:print t**. Here is the result:

```
* (load "foo.lsp" :print t)
; Reading file C:\GCLISP\FOO.LSP

5

ERROR:
End of file while reading s-exp.
1> _
```

The error message means "an end-of-file was found while reading an s-expression". That is: the end of the file was read before finding the close parenthesis needed to complete the form in process.

Only the first LISP form in the file returned a value, before the error message appeared. This says that the error must be in the second form, and the evaluation halted there (otherwise the return value for the second form would have printed).

With a small file like this one, there is no real need to use the :print option; but the option is very useful when reading a large file.

1.11 Table of COMMON LISP Language Conventions

The following table describes briefly several of the language conventions found in COMMON LISP (you have encountered some of them in this chapter). For more complete discussion of these and other conventions, see Chapter 1 of the GCLISP Reference Manual and Chapter 1 of the COMMON LISP Reference Manual.

<table>
<tr><th>CONVENTION</th><th>MEANING</th></tr>
<tr><td>()</td><td>Parentheses demarcate a list. The GCLISP listener interprets a list as a function call, a macro call, or a special form.</td></tr>
<tr><td>'</td><td>A single quote indicates that the form that follows is not to be evaluated. 'form is the same as (quote form).</td></tr>
<tr><td>;</td><td>A semi-colon is the comment character. Any data to its right (on an input line) will be ignored by the input reader. In output, anything to the right is an informational message.</td></tr>
<tr><td>"</td><td>Double quotes enclose character-string data: "This is not 39 characters long".</td></tr>
<tr><td>\</td><td>The character following the backslash character is accepted literally by the input reader, without any special meaning. (For example, all of the special characters in this table, including backslash, lose their special meanings when preceded by backslash.)</td></tr>
<tr><td>|</td><td>Vertical bars appear on either side of a symbol name or around characters in the symbol name to mark special characters for treatment as literal characters.</td></tr>
<tr><td>:</td><td>A colon associates a symbol name with the package it belongs to.</td></tr>
</table>

Chapter 2

The GMACS Editor

GMACS is a full-screen display editor modeled after EMACS, the editor created by Richard M. Stallman at the MIT Artificial Intelligence Laboratory.

You can scan the quick-reference command summary in section 2.8 below to see that GMACS is a modern full-featured text editor, with a repertory of nearly one hundred commands bound to keychords and short key sequences. The kinds of objects which can be manipulated by these commands include characters, words, character strings, lines, arbitrary user-defined regions of text, edit windows, edit buffers, and files.

The particular strength of GMACS, however, is that it implements commands and features for editing LISP code. Using these, you can manipulate all of the important elements of LISP -- s-expressions, lists, lines of code, comments, and function definitions -- as well as controlling interactively the appropriate indentation and parenthesizing of your LISP expressions.

Section 2.7.7 summarizes the LISP-editing commands, and section 2.4 describes these commands and features in more detail. The GMACS LISP-editing features will:

- automatically blink the open parenthesis which matches the current close parenthesis;

- inform you when you have typed too many close parentheses;

- indent an s-expression or a line correctly;

- move forward and backward over s-expressions, and cut and paste them;

- display the parameter list or detailed documentation of either an interpreted function or a compiled function; or display the expansion of a macro form;

- display the full name and documentation of a GMACS command or a LISP function when you remember only part of the name;

- exit to a temporary LISP listener and re-enter GMACS without disturbing your GMACS environment (using only one short command for an exit or re-entry);

- evaluate directly, using the temporary listener -- and without leaving GMACS -- the LISP statements which you have been typing into the GMACS edit buffer.

An on-line tutorial in the use of GMACS can be invoked from within GMACS using the keychord **Alt-H T** for "Help teach" (see section 2.1.8).

This chapter as a whole describes the various features of GMACS and explains how to use them:

- Section 2.1 gives you basic information for getting started with GMACS.

- Sections 2.2 through 2.4 give more detailed explanations of the various facilities of GMACS:

  * Section 2.2 deals with commands and capabilities for manipulating edit buffers and the associated files.

  * Section 2.3 describes GMACS capabilities and commands for general editing.

  * Section 2.4 is concerned with the set of commands specifically designed to manipulate LISP programming language constructs.

- Sections 2.5 through 2.8 provide reference listings for GMACS commands:

  * Section 2.5 lists the commands bound to the function keys on the IBM PC keyboard.

  * Section 2.6 lists the commands bound to the cursor motion keys on the IBM PC keyboard.

  * Section 2.7 provides a summary reference to all GMACS commands, including their key bindings and a short description of each command.

  * Section 2.8 is a quick-reference listing of all GMACS commands and their key bindings.

2.1 The GMACS Environment

2.1.1 Entering GMACS

You have two ways of entering the editor from your GCLISP environment:

1. using the **Ctrl-E** keychord, which has the same effect as the function call **(ed)**; or

2. using the **ed** function in one of these forms:

   (ed "<filename>")
   (ed t)

When you first invoke GMACS with no filename (the form **(ed)**), you are placed in an empty edit buffer (see section 2.1.4) called "MAIN". If you specified a filename, then the contents of that file are read from disk into a buffer named after the file. The form **(ed t)** gives you a new empty MAIN buffer (and preserves the MAIN buffer from a previous invocation, if any).

When you invoke GMACS for the first time in any GCLISP session, the editor programs must be loaded into your computer's memory, to establish the GMACS environment. The time required to load the editor will vary with your computer system. Your screen will display a message line showing the progress of the loading process.

2.1.2 Exit and Re-Entry

To leave GMACS and return to the interpreter environment, type the key sequence **Ctrl-X Ctrl-C**.

When you again invoke GMACS, via **Ctrl-E** or the **ed** function, the GMACS environment of buffers and files will be re-established. If your command is **(ed)**, without a filename, you will be placed in the buffer where you were last editing, and at the same point in that buffer. If your command is **(ed "<filename>")**, then GMACS will re-establish the edit environment following the rules of the FIND-FILE command (see section 2.2.5, "Reading a File").

2.1.3 Protecting the Buffer Contents

At any time in a GMACS session, several edit buffers may exist. The set of existing buffers is preserved in the GCLISP workspace when you exit from GMACS. These will all be available to you when you re-enter GMACS from the GCLISP interpreter environment. Their contents will be exactly as you left them.

However, when you are editing a buffer, you should write out the buffer to the file often. There are good reasons for this. In any of the following circumstances, the contents of the GCLISP workspace, including the buffers, are irretrievably lost:

- when you exit from GCLISP;
- when the operating system or GCLISP has to be re-initialized due to some unforeseen problem; or
- when the computer is turned off.

The commands for handling buffers and files are found in section 2.2 below, "Manipulating Buffers and Files".

2.1.4 Buffer, File, Window, and Screen

Four things are central to learning how editing is done in GMACS. This subsection presents these concepts, to avoid any possible confusion.

The four things are:

- the edit buffer
- the file being edited
- the edit window
- the edit screen

A very brief explanation of the roles of these four is as follows:

> The *edit screen* is the entire terminal display screen during a GMACS editing session. The most important area on the edit screen is the *edit window*. In this window is displayed (part or all of) the contents of the *edit buffer*. These contents often consist of a working copy of a *file being edited*.

Now for details.

The edit buffer.

This is a temporary storage area for lines of text being edited. The area -- sometimes called just "the buffer" -- is in the GCLISP workspace. It is maintained by GMACS during an editing session.

When in GMACS, there is at any particular time just one particular buffer where editing occurs, the *current buffer*. Strictly speaking, editing consists of changing the contents of the edit buffer by adding or deleting characters at particular places. You may type individual characters, or words, or LISP forms, into the buffer; or manipulate the buffer contents by rearranging, copying, or deleting larger blocks of text. But it all comes down to changing the character-by-character contents of the edit buffer. So we speak of "editing the buffer", or being "in" the buffer.

At any particular time, the buffer may be empty. Or it may contain lines you have typed in; or a copy of a file on disk that was read into it; or any combination of lines originally gotten either from a disk file or typed in by you.

The buffer is an object that you can manipulate. You can create one or delete it; or give it a name; or read a file into it; or make another existing buffer the current edit buffer. The set of GMACS operations on buffers is described in section 2.2 below, "Manipulating Buffers and Files". For now, though, the important fact about the edit buffer is that this is where editing happens.

The file being edited.

A file is a named storage area in a directory on a disk in your computer. Once created, it stays there until you delete it explicitly, with an operating-system command such as del *filename*. You may type a file to the terminal screen, or print it (if text), or copy it, or merge it with other files, or delete it, etc.

You can also edit a file with GMACS. Strictly speaking, though, the file itself is not edited. Only the contents of an edit buffer can be edited; and a file is not an edit buffer.

To edit a file, you give a GMACS command to "read the file into the edit buffer". This means: find and open the file on the disk, and read its contents into the edit buffer. This is a copying operation, and has no effect at all on the contents of the file as stored on the disk.

Then you edit the copy in the buffer.

Finally, if you are satisfied with your changes, you give a GMACS command to "write the file". This means: write the contents of the buffer to the disk and give this disk file the same name as before. In the process, the old, unchanged copy of the file on the disk is automatically deleted. This writing must be done in order to save permanently the results of the editing, since the buffer itself goes away when GCLISP does.

Thus:

- We say "edit the file", but the actual changes are made on the copy of the file that has been read into the buffer.

- The file is permanent. The buffer contents are not.

- Reading the file (from disk) into the buffer has no effect on the file contents. Writing the buffer to the file (on disk) replaces the old version of the file with the edited version.

The edit window.

This is an area on the terminal display screen. It provides a view into the edit buffer. In the edit window are displayed as many lines of the current contents of the edit buffer as will fit there.

At any particular time, you can edit only the part of the buffer currently showing in the window. That is, text can be inserted or deleted only at a point in the part of the buffer currently showing in the edit window.

The edit screen.

This is the terminal display screen as it is presented while you are in GMACS. GMACS controls your use of the screen, dividing it into several areas. The most important area is the edit window. Other areas are the mode line and the echo/message area. The screen is divided into these three areas, except when the type-out window may obscure the edit window.

More details about the edit screen follow.

2.1.5 The Edit Screen

This sample screen shows the display after you initially enter GMACS (without specifying a filename at entry). The screen is empty except for the line of information near the bottom and the cursor mark in the upper left-hand corner.

```
|
| _
|
|
|
|
|
|
|
|
|
|
|
|
| GMACS V1.00  MAIN: null pathname              Alt-H = HELP
|
|_____________________________________
```

A *mode line* like the one above is a permanent feature of the edit screen. It always displays as the third-from-last line on the screen. Reading left to right, it has these elements:

- The editor name and version number: "GMACS V1.00";
- The name of the current edit buffer, e.g. "MAIN", followed by a colon;
- The pathname of the file currently being edited in this buffer, or "null pathname" if no file is associated with this buffer;
- An asterisk (*, the *buffer-status*), if the contents of the current edit buffer have been changed since they were last written out to, or read from, a file;

- "Alt-H = HELP", to remind you how to invoke on-line GMACS help.

In most sample screens appearing in this chapter, the mode line will be omitted; it is usually unnecessary for understanding the point being made.

The space above the mode line is usually filled by the edit window, also known simply as "the window". (Other windows will always be specifically identified.) The edit window provides a view of the current edit buffer: either all of the buffer, or as much of it as can be displayed in the window area on the screen. You can edit data already in the buffer, or type in new data, only in the area of the buffer currently displayed in the window.

As data is typed into a buffer, the buffer expands automatically to hold it. The cursor moves as you continue to type. It shows where the next character typed will be inserted into the buffer. Character insertion (or deletion) always occurs at the point (called the *point*) between the character above the cursor and the character immediately preceding it. Note that a "non-printing character" such as a space, a tab, or a newline is like any other character in this regard. For example, the newline character (produced by <ENTER>) doesn't show in the screen display; but it is in the buffer like any other typed data.

When the data in the buffer fills the edit window, the window shifts down so that you can continue to see what you type into the buffer. To review and edit what you have typed, you can move the window back and forth across the buffer (see sections 2.3.3 - 2.3.4 about the cursor motion commands). An entire buffer of any size can be viewed in this way, one window at a time.

2.1.6 A GMACS Glossary

Here is a short glossary of the most important terms for the various elements of the GMACS screen image and the related edit buffer.

EDIT WINDOW A part of the terminal display screen used for the purpose of displaying the contents of the edit buffer. The edit window usually occupies all but the bottom three lines of the display screen.

EDIT BUFFER A temporary storage area created and used by GMACS. (The area is in your GCLISP workspace.) The *active* or *current* edit buffer appears in the edit window.

CURSOR and POINT
The *cursor* appears as a blinking mark (usually an underline or a rectangle) on the edit screen. The *point* is a position in the current buffer: the position between the cursor and the preceding character position. Thus, if the cursor is under the letter "a" in the word "bar" (bar), the point is between "b" and "a". Deletions and insertions in the buffer take place at the point.

ECHO AREA/MESSAGE AREA
The bottom two lines of the edit screen. Here, edit commands that you type are displayed ("echoed"); this enables you to easily verify your input commands. Miscellaneous informational messages appear here also.

MINI-BUFFER An area where you are prompted to enter the names of files and other information required by certain commands. The mini-buffer appears in the right half of the echo area.

MODE LINE The line of status information appearing near the bottom of every edit screen. The mode line displays the name of the editor (GMACS) and its version number; the name of the current buffer and the associated file, together with the buffer-status; and the Help keychord.

BUFFER-STATUS The condition of the buffer with respect to changes. If you have added or deleted data in the current edit buffer since last reading in a file to the buffer, or writing out the buffer to a file, an asterisk appears following the filename in the mode line. Otherwise this space is blank.

TYPE-OUT A display of information produced by a GMACS command. It appears in the *type-out window*, a temporary window in the top part of the edit screen. (The type-out window temporarily overlays part or all of the edit window.)

2.1.7 Inputting Commands and Characters

While you are in the GMACS environment, everything you type at the keyboard is part of an *edit command*. An edit command directs GMACS to perform an editing task. (The edit commands are actually LISP functions.)

An edit command is invoked by typing an alphanumeric key, a *keychord*, a *key sequence*, or a special function key. A key or keychord or key sequence that invokes a command is said to be *bound* to the command, and vice versa.

Most of the alphanumeric keys on the keyboard -- the alphabetic keys, the numeric keys, and the punctuation keys -- are bound to an edit command that inserts the character represented by the key into the edit buffer. In other words, typing the key **A** has the same effect as a command "insert the character **A**".

A keychord consists of a *modifier key* and an alphanumeric key. The modifier key must be held down while the alphanumeric key is pressed. The modifier keys are the shift key, and the **Ctrl** key and the **Alt** key, both located just left of the alphabetic keys on the PC keyboard. (The shift key is used mainly for inserting upper-case letters.)

A keychord is represented in print by the symbols of the appropriate keys linked together with hyphens. The printed form **Ctrl-F** indicates that the **Ctrl** key is held down while the **F** key is pressed.

A key sequence consists of either a keychord followed by an alphanumeric key, or else a keychord followed by another keychord. The additional key or keychord is pressed after the keys for the first keychord have been released.

A key sequence is represented by the keychords and keys written one after the other. The printed form **Ctrl-X 2** indicates that the **Ctrl** key is held down while the X key is pressed, and then -- after the keychord is released -- the **2** key is pressed.

For convenience, a number of edit commands bound to keychords or key sequences have also been bound to the function keys on the IBM PC keyboard. To invoke one of these commands, you do not have to use the keychord or key sequence, but can use the function key instead. (See section 2.5, "Table of Function Keys.")

A few other special keys -- the cursor motion keys, **Rubout**, **Home** and **End**, and **Delete** -- invoke editing actions in GMACS, rather than representing characters for insertion into the buffer (unless they have been shifted by the **NumLock** key to implement the numeric keypad).

2.1.8 GMACS Help

At any time while in the GMACS environment, you can invoke on-line help about GMACS.

To invoke the GMACS on-line help facility, type **Alt-H**. (This keychord always appears at the right-hand end of the GMACS mode line.) This invokes the command HELP-DEADEND, which displays in the mini-buffer a short menu of options and how to invoke them:

```
GMACS V1.00  MAIN: null pathname            Alt-H = HELP
Help    ?=Help guide  D=Document command  T=Teach GMACS
        A=Apropos     K=Keychord binding
                             Please enter your selection:
```

The ? option invokes the help guide, a display of more detailed descriptions of the options:

```
These kinds of GMACS on-line help are available.
To invoke one of them, type Alt-H followed by A, D, K, T, or ?.

 A   "Apropos" - Displays the keychords for all GMACS
     commands that contain a specified string.
     Prompts for the string.

 D   "Documentation" - Displays documentation on all
     GMACS commands containing a specified string.
     Prompts for the string.

 K   "Keychord binding" - Displays the GMACS command bound
     to a specified keychord.  Prompts for the keychord.

 T   "Teach GMACS" - Invokes the GMACS on-line tutorial.

 ?   Displays this guide.
```

The help guide appears in a type-out window.[8] So does the Help information which is displayed when you request it via one of the listed options. To invoke one of the options, type the

----

8. Note that when a type-out window has been displayed, you are prompted to type a space character to continue. Use the space bar, because any other input will be executed as a GMACS command. For example, any self-inserting character will be inserted into the current edit buffer.

appropriate key (A, D, K, T, or ?) at the prompt. For rapid access to an option, you can invoke it directly by a key sequence without waiting to see the menu:

Alt-H A ED-APROPOS
Prompts you for a character string, and displays in a type-out window all GMACS commands which contain in their name the specified string.

Alt-H D ED-DOC
Prompts you for a character string, and displays in a type-out window the on-line documentation for all GMACS commands which contain in their name the specified string.

Alt-H K ED-KEYCHORD
Prompts you for a keychord, and displays in a type-out window the command associated with the specified keychord.

Alt-H T ED-TEACH
Invokes the GMACS on-line tutorial.

Alt-H ? ED-HELP
Displays the help guide consisting of descriptions of the options listed in the help menu.

2.1.9 Aborting GMACS Commands

It is sometimes convenient to abort an editing command, rather than letting it complete. Two special GMACS commands let you do this.

Esc DEADEND
The **Esc** key ("escape") aborts the current command, rings the terminal bell, and returns you to normal GMACS command entry.

Ctrl-G ED-BEEP
This command aborts the current command, rings the terminal bell, and returns you to normal GMACS command entry.

Note that **Esc** has other meanings in certain other GMACS commands. **Esc** operates as DEADEND except in these specific cases (described in the documentation of the particular commands elsewhere in this chapter).

2.2 Manipulating Buffers and Files

The edit buffer and the file being edited were described in section 2.1.4, "Buffer, File, Window, and Screen". The current section summarizes the relation between buffer and file, and describes the GMACS commands for manipulating buffers and files.

When you have finished editing in a buffer for the time being, you can copy ("save" or "write") the contents of the buffer to a disk file for more permanent storage. To modify an existing file, you can copy ("read") the file into an edit buffer.

When you edit an existing file, you edit only the copy of it that has been read into the buffer. If you decide not to keep the changes you make while editing, you can delete the buffer instead of returning it to disk storage. If you want to keep both the earlier version of the file and the newly edited version, you can write the new version to disk with a new name and it becomes a separately stored file.

The commands for all of these operations are described below.

2.2.1 How Buffers and Filenames are Related

When you read a disk file into a buffer, or when you write out the contents of a buffer to a file, GMACS associates the file and the buffer by name.

At any time, the filename currently associated with a buffer is the name of the file most recently read into or written out from the buffer. This name changes only when you specify another filename for reading from or writing to.

This association is maintained by GMACS during your editing session (and even between sessions, as described in section 2.1.2 above). You can see the complete list of names of your existing buffers and the filenames associated with them by using the LIST-BUFFERS command described in the next section.

If GMACS has newly created a buffer and the buffer is empty, then there is no file associated with the buffer, and the designation "null pathname" appears in the mode line.

2.2.2 Displaying Buffer Names

Because buffers stay in GMACS until you delete them, you may need to know what buffers currently exist. You may also need to know whether the contents of a buffer have been written out to a file after the most recent changes made to the contents.

To find out these things, use the LIST-BUFFERS command, invoked with the key sequence **Ctrl-X Ctrl-B**. This command lists (in a type-out window) the name of each edit buffer and the name of the file associated with it.

An asterisk (*) appears next to the filename if the buffer has been modified since it was last saved or written to disk with a SAVE-FILE or a WRITE-FILE command, as described below.

2.2.3 Marking a Buffer Unmodified

To mark a buffer unmodified, use the command **Ctrl-X U**. This directs GMACS to regard the buffer contents as having been unchanged since the most recent READ, WRITE, or SAVE of the buffer contents. In response to the command, GMACS clears the buffer-status (*) in the mode line. (You would use this command when you edit a buffer, modifying its contents, and then decide that you do not want to save the changes; or when you change the buffer contents by a typing mistake.) Note that the modifications are not undone by this command. The only action GMACS takes is to clear the buffer-status.

2.2.4 Selecting a New Current Buffer

Recall that the *current buffer* is where editing is done at any given time. There is always a current buffer.

To select a different buffer to be the current buffer, use the SELECT-BUFFER command, invoked by pressing **Ctrl-X B**. This command prompts for the name of the buffer to switch to.

The command SELECT-PREVIOUS-BUFFER (**Ctrl-X P**) selects the buffer in which you were last editing before entering the current buffer.

2.2.5 Reading a File

To read a specific file into some buffer other than the current buffer, or into a new edit buffer, use the FIND-FILE command, executed with the key sequence **Ctrl-X Ctrl-F**. The command prompts you for the filename of the desired file.

If a buffer exists that is associated with this file, it is selected as the current buffer and nothing is read into it. The point is positioned where it was last located when that buffer was last the current buffer.

Otherwise, GMACS looks among the existing buffer names for a buffer named after this file. (When a buffer is named after a file, the buffer name is the name of the file without the "extension" part, if any, of the filename. By this rule, a buffer would be named CONSOLE for either the file CONSOLE.CON or CONSOLE.LSP.)

If you have specified FIND-FILE for the file CONSOLE.CON and a buffer named CONSOLE is already in use but CONSOLE.CON is not associated with the CONSOLE buffer, then GMACS will create a new buffer named CONSOLEX and read CONSOLE.CON into it. In other words, a new buffer will be created for the requested file, and its name will be the filename with an "X" appended (and without the extension).

If no buffer is associated with the filename, and there is no buffer named after the file, then the command creates a new buffer named after the file and reads the file from disk into the new buffer.

To read a specific file into the *current* buffer, use the READ-FILE command, invoked with the key sequence **Ctrl-X Ctrl-R**. You are prompted for the name of the file to read.

Whatever is already in the current buffer is written over (lost) by the reading in of the file. If you have made changes to the current buffer since you last wrote it to disk (via SAVE-FILE or WRITE-FILE), READ-FILE warns you and offers the option of cancelling the command.

Note that, as a result of this behavior, FIND-FILE is a safer command than READ-FILE. READ-FILE will destroy the current contents of an existing, unmodified buffer without warning you, while FIND-FILE will not destroy the current contents of any buffer.

2.2.6 Writing a File

After you have edited a file in a buffer, or typed text into an empty buffer, you transfer the buffer's contents to a disk file (unless you decide not to save the editing you have done).

To put the buffer's contents to a file, use either the SAVE-FILE command or the WRITE-FILE command. SAVE-FILE is executed with the key sequence **Ctrl-X Ctrl-S**. This command writes the contents of the buffer to a file with the name currently associated with the buffer. This replaces the old version of the file with the new, edited version. If the buffer has not been associated with a disk file, you will be prompted to name a file where you want to save the contents of the buffer.

If you do not want to replace an existing file with the contents of the buffer, use the WRITE-FILE command, executed with the key sequence **Ctrl-X Ctrl-W**. This command prompts you for a filename and writes the contents of the buffer as a file with the new filename.

2.2.7 Deleting a Buffer

To eliminate a buffer, use the KILL-BUFFER command, invoked with the keychord **Ctrl-X K**. This command prompts you for the name of a buffer and erases the buffer with that name. If you press the ENTER key without entering a buffer name, the command deletes the current buffer and returns you to the previous buffer.

If the buffer has been modified since it was last written to a file, you will be asked to verify the KILL-BUFFER operation. If you decide not to complete the command, press **Ctrl-G**.

2.2.8 Directory Operations

While in GMACS, you can read or write files in the working directory. You may want to change the working directory; or you may want to examine the contents of this directory or of some other directory. The following two commands enable you to do that.

Ctrl-X Ctrl-D DISPLAY-DIRECTORY
Use this command to obtain a listing of the names of files in any particular directory. You are prompted for the pathname of the directory you want. You can specify either a directory or a filename, or a set of filenames using the "*" wild-card convention, just as in the DOS dir command. The directory listing is displayed in a type-out window.

Ctrl-X C CHANGE-DIRECTORY
Use this command to change the working directory to the directory you name in response to the prompt displayed following this command.

2.3 Editing Text

All of the GMACS commands in the following subsections are useful for editing general text files. The many commands designed specifically to edit LISP forms are described in section 2.4 and its subsections.

2.3.1 Inserting and Deleting Text

The simplest editing consists of inserting and deleting individual characters in an edit buffer.

You insert single characters by typing the character keys on the keyboard. As you press individual keys, the characters they represent are entered into the buffer one after the other.

The edit window shows the results, character by character. The point moves along as you type. The cursor is always one character position ahead of the character that was last typed. If there are characters in the buffer ahead of the point, they are shifted one character ahead with every new character inserted.

To erase a character you have just typed, press the **Rubout** key. This is the key labeled with a left-pointing arrow (<--), located in the top row of keys on the IBM PC keyboard, just northeast of the alphabetic keys.

For example, here is a line before and after typing the **Rubout** key:

```
LISP is the language of AE_

LISP is the language of A_
```

Here, the underlines show the before-and-after cursor positions.

To delete the character at the cursor position (rather than the preceding character), invoke the DELETE-CHAR command by pressing **Ctrl-D** or **Del.** The character at the cursor disappears, and all characters following the cursor move one character backward.

Two special-purpose commands can be used to delete extra spacing in the text:

Ctrl- DELETE-HORIZONTAL-SPACE
This deletes any spaces or tabs adjoining the point on either side.

Ctrl-^ DELETE-INDENTATION
This deletes any indentation at the beginning of the current line, and the preceding newline character. This action appends the current line to the preceding line.

2.3.2 Words and Lines

Many GMACS commands specify an operation on a *word* or on a *line*. You need to know exactly is meant by a word or a line in order to use the commands effectively.

To GMACS, a *line* consists of the sequence of characters from one newline character to the next (including the ending newline). There may be more characters in this line than can fit in a single line of the display screen. Then more than one display line will be used to display the line.

Such a line in the edit buffer is called a *wrapped line* on the display, because the line "wraps around" the end of one display line and continues on the next. GMACS informs you that a display line is wrapped by placing an exclamation mark (!) in the right-most display position:

```
|
| This line wraps onto the next line and the!
| next line wraps onto the line after it. Th!
| ere is no newline character in the text._
|
|____________________________________
```

To GMACS, a *word* is any string of *alphanumeric* characters: that is, letters or digits. So the end of a word is marked by any other character: a punctuation symbol, any other special character, or *white space*: a space, tab, or newline character.

When a GMACS command specifies an operation on a "word", such as FORWARD-WORD, it means that the operation should be applied to the nearest string (in the correct direction) which satisfies this meaning. Thus, FORWARD-WORD means: find the first alphanumeric character in the forward direction, and place the point at the end of the "word" that begins with that character.

2.3.3 About the Cursor Motion Commands

These commands enable you to move the point around in the edit buffer. This is needed when you want to make insertions or deletions somewhere other than the current point, or to view some other part of the edit buffer.

There are commands to move the cursor over a character, a word, a line, a screen, or an entire buffer. The commands come in pairs: for each unit of movement, one command moves the cursor forward and one command moves it backward. (For lines, there are two pairs of commands; see below.)

When the point is already at one end of the window and a cursor motion command attempts to move it "off the end", the window will be *scrolled* -- moved over the edit buffer -- so that the needed new area of the edit buffer appears in the window and the point moves as desired.

The NEXT-LINE (**Ctrl-N**) and PREVIOUS-LINE (**Ctrl-P**) commands move the point up and down in the edit window by one buffer line. The point moves up or down the window in the same column where it began; but when a shorter line is encountered, the point moves to the end of the line. If a line is wrapped, a NEXT-LINE, PREVIOUS-LINE, BEGINNING-OF-LINE, or END-OF-LINE command may move the cursor over several display lines.

The commands END-OF-BUFFER and BEGINNING-OF-BUFFER set the current mark (see section 2.3.9). This behavior enables you to return quickly to where you were before giving the command.

2.3.4 Table of Cursor Motion Commands

The following list summarizes the cursor motion commands. It also lists the cursor motion keys, or keychords involving these keys, which will execute these commands.

The command keychords should be used in preference to the cursor-motion keys or keychords. The keys that make up a command keychord are closer to the usual position of your hands centered in the keyboard, while the cursor motion keys are off to the right. Once the editing commands are familiar to you, typing will be faster if you use the command keychords. Over the course of many repetitive editing operations, this will save time.

Ctrl-F or Right Arrow
: FORWARD-CHAR
  Moves the cursor to the right (forward) one character.

Ctrl-B or Left Arrow
BACKWARD-CHAR
Moves the cursor to the left (backward) one character.

Alt-F or Ctrl-Right Arrow
FORWARD-WORD
Moves the cursor forward one word.

Alt-B or Ctrl-Left Arrow
BACKWARD-WORD
Moves the cursor backward one word.

Ctrl-E
END-OF-LINE
Moves the cursor to the end of the current line.

Ctrl-A
BEGINNING-OF-LINE
Moves the cursor to the beginning of the current line.

Ctrl-N or Down Arrow
NEXT-LINE
Moves the cursor to the next line (down one).

Ctrl-P or Up Arrow
PREVIOUS-LINE
Moves the cursor to the previous line (up one).

Ctrl-V or PgDn
NEXT-SCREEN
Moves the window forward in the edit buffer by about one window-length (one edit screen). The window is positioned on the edit buffer so that the previous last line in the window becomes the new first line. (This makes it easier to locate yourself for editing in the new window.)

Alt-V or PgUp
PREVIOUS-SCREEN
Moves the window backward in the edit buffer by about one window-length (one edit screen). The window is positioned in the edit buffer so that the previous first line in the window becomes the new last line.

Ctrl-L
REDISPLAY-SCREEN
This command redisplays the entire screen so that the current line is near the middle of the edit window. Given a number *n* as argument, the current line will be the *n*th line from the top in the redisplay if *n* is positive, and *n*th from the bottom if *n* is

negative.

Ctrl-Z > or End
END-OF-BUFFER
Moves the cursor to the end of the buffer.

Ctrl-Z < or Home
BEGINNING-OF-BUFFER
Moves the cursor to the beginning of the buffer.

2.3.5 Inserting New Lines

You can insert a new line of text with the OPEN-LINE command, executed with **Ctrl-O**. This command inserts a newline character at the point, and leaves the point before the newline character:

```
      (before Ctrl-O)            (after Ctrl-O)

      |                          |
      | line one                 | line one
      | line_two                 | line_
      | line three               |  two
      |                          | line three
      |__________________        |__________________
```

If you are in the middle of a line and want to add text, use **Ctrl-O**.

If you are at the end of a line and want to continue with another line, use the ENTER key. This inserts a newline character at the point, and leaves the point at the beginning of the new line:

```
   (before <ENTER>)                 (after <ENTER>)

   |                                |
   | line one                       | line one
   | line_two                       | line
   | line three                     | _two
   |                                | line three
   |___________________             |___________________
```

2.3.6 Numeric Arguments (Repeat Counts)

You will often want get the effect of executing a GMACS command a certain number of times one after the other. For example, you may want to move the cursor forward exactly 65 characters. It would be a nuisance to repeat a cursor-motion command this often. Instead, you can invoke the single command with a *numeric argument* which specifies how often the command is to be repeated.

To do this, precede the command with the key sequence:

Ctrl-U <number>

That is: type **Ctrl-U**, then the numeric argument, and then the command. In this context, the number is called the *repeat count* for the command which follows it.

For example, to advance the cursor 65 characters:

Ctrl-U 65 Ctrl-F

Ctrl-U alone, without a numeric argument specified, performs the command 4 times. In other words, there is a "default repeat count" of 4. To advance the cursor 4 characters:

Ctrl-U Ctrl-F

Any additional **Ctrl-U** which *follows* the repeat count argument multiplies the repeat count by 4. This input advances the cursor by 64 characters:

Ctrl-U 16 Ctrl-U Ctrl-F

Since the default repeat count is 4, this input does the same:

Ctrl-U Ctrl-U Ctrl-U Ctrl-F

That is: the two "extra" **Ctrl-U** keychords multiply by 16 the default repeat count of 4.

Since a repeat count can result in a large change in the buffer contents, it's important to type the key sequence with care -- especially the value of the repeat count. To help you verify your typing, the value of the count appears in the form **<number>:** in the echo area as you type it.

Remember that the ordinary characters of the keyboard are self-inserting input: typing the character **A** means "insert the character **A**". Thus, to insert a row of 65 asterisks into the buffer:

Ctrl-U 65 *

With some commands, the **Ctrl-U** prefix causes different behavior unrelated to a numeric argument. This behavior will be made explicit in the descriptions of the particular commands.

2.3.7 Setting Upper-Case and Lower-Case

To aid you in formatting text, GMACS has commands for setting the case of alphabetic characters to upper-case (capitals) or lower-case (small letters).

The first three following commands are convenient for setting the case of a word to "initial-caps", "all-small", or "all-caps". The other two commands set the case of an entire region (see section 2.3.9, "Manipulating Regions and Marks").

KEY	**COMMAND**
Alt-C	UPPERCASE-INITIAL Capitalizes the letter (if any) following the point, and lower-cases the rest of the word. (that is, initial-caps the word starting at the cursor).
Alt-L	LOWERCASE-WORD Lowercases the word starting at the cursor.
Alt-U	UPPERCASE-WORD Uppercases the word starting at the cursor.
Ctrl-X Ctrl-U	UPPERCASE-REGION Puts all the letters in the current region in upper-case.
Ctrl-X Ctrl-L	LOWERCASE-REGION Puts all the letters in the current region in lower-case.

2.3.8 Search and Replace Commands

You often need to locate a particular character string, for example a particular word, within a text. You may want to delete the word, or replace it with another, or do other editing at that location. You may want to do this at only one instance of the word; or at every instance of the word; or at selected instances of the word.

The FORWARD-SEARCH, REVERSE-SEARCH, QUERY-REPLACE, and GLOBAL-REPLACE commands make these operations easy:

Ctrl-S	FORWARD-SEARCH Repositions the point at the next instance of a character string that you specify.
Ctrl-R	REVERSE-SEARCH Repositions the point at the preceding

instance of a character string that you specify.

Alt-% or **Alt-5**

QUERY-REPLACE
Finds every instance of the string between the point and the end of the buffer; and allows you to selectively replace each such instance with another pre-specified string.

Alt-* or **Alt-8**

GLOBAL-REPLACE
Replaces every instance of the string between the point and the end of the buffer, with another pre-specified string.

Thus, QUERY-REPLACE and GLOBAL-REPLACE can perform actual editing in the buffer. FORWARD-SEARCH and REVERSE-SEARCH only reposition the cursor to a place where you want to edit.

When any of these commands is given, you are prompted (in the mini-buffer) to enter the search string. The commands are not case-sensitive to the search string you specify: a search for "LISP" will also find "Lisp" and "lisp".

Each of the commands automatically re-positions the edit window as necessary to show the located string. However, if the command finds no instance at all of the specified search string, the cursor is not moved from its original position. Also, when QUERY-REPLACE or GLOBAL-REPLACE has searched to the end of the buffer (whether it finds instances along the way or not), the cursor is returned to its original position. (This happens also if you abort QUERY-REPLACE.)

When QUERY-REPLACE finds an instance of the string, it halts and prompts you with four options. Your choices are:

- type **Y** (to replace that instance and continue searching)

- type **N** (to leave that instance unchanged and continue searching)

- type **!** (to replace all remaining instances to the end of the buffer, without further prompting)

- type **Ctrl-G** (to abort the command -- no more searching or replacing)

2.3.9 Manipulating Regions and Marks

The editing operations described so far have included insertions on characters, words, and lines. These are natural units to manipulate with the editor. Often, however, it's convenient to manipulate larger blocks of text: to move, copy, or delete paragraphs or other large units.

GMACS enables you to define and manipulate text in blocks of any size, called *regions*. Unlike a character or a word or a line, a region is not "naturally" defined: it is not delimited by blanks or newlines, for example. The limits of a region are completely up to you.

You specify one end of a region by moving the cursor there and then giving the SET-POP-MARK command (**Ctrl-@**). This sets a *mark* at the point. The mark doesn't show in the edit window; but the message "Mark set" appears in the message area.

To specify the other end of the region, move the cursor there (either backward or forward from the mark). Then execute the command to do the desired particular operation on the region, which consists of the area of the buffer between the mark and the point.

Three basic operations can be performed on a region:

- A case operation, already described in section 2.3.7

- The command KILL-REGION (**Ctrl-W**)

- The command SAVE-REGION (**Alt-W**)

The KILL-REGION and SAVE-REGION commands are useful in deleting, copying, or moving the contents of the region (see section 2.3.10, "Killing and Recovering Text").

You can also specify a sequence of marks for immediate or later use. GMACS keeps a list of these, the *mark pdl* -- "pdl" for "push-down list". You can add a mark to this list; throw away a mark from the list; or recover and use a mark which is currently on the list. You should think of the list as a stack of marks, which you manipulate with the following commands:

Ctrl-@ SET-POP-MARK
The command SET-POP-MARK defines a mark (at the current location of the point); and puts the mark on the top of the stack. Each mark already on the stack is "pushed down": the top mark becomes the second, the second becomes the third, and so on. The top mark is also

called the *current mark*.

Ctrl-U Ctrl-@ SET-POP-MARK
This "gives you the top mark": it gets the current mark and places the point at that position. The mark is taken off the stack. All the remaining marks, if any, are moved up one: the former second mark is now the current mark, etc.

Ctrl-U Ctrl-U Ctrl-@
SET-POP-MARK
This command takes the current mark off the stack without placing the point at the mark. All the remaining marks, if any, are moved up one.

These three commands enable you to define, store, recover, and delete marks whenever you like. Besides using a mark to delimit a region, you may want to use a mark simply as a way to mark a point in the buffer to which you will want to return at some later time for further editing.

One additional command enables you to move the point quickly to the current mark, without changing the region and without discarding the mark:

Ctrl-Z Ctrl-X or **Ctrl-Z Space**
EXCHANGE-POINT-AND-MARK
This exchanges the point and the current mark.

The cursor motion commands END-OF-BUFFER and BEGINNING-OF-BUFFER (see section 2.3.4) set the current mark. This behavior enables you to return quickly to where you were before giving the command.

2.3.10 Killing and Recovering Text

In section 2.3.1, you met the DELETE commands **Ctrl-D** and **Rubout**, which operate on individual characters. Text deleted from the buffer with one of these commands is not saved anywhere; so it can't be recovered.

All other commands that remove text, the "kill commands", save the deleted text so that it can be recovered. GMACS maintains the special area where the deleted text is saved; it is called the *kill history*.

The kill commands operate on words, lines, and regions. This is the set of kill commands:

Alt-D	KILL-WORD Moves the current word (from the point forward to the end of a word) to the kill history.
Ctrl-Rubout	BACKWARD-KILL-WORD Moves the current word (from the point backward to the beginning of a word) to the kill history.
Ctrl-K	KILL-LINE Moves to the kill history the text forward from the point to the end of the current line, excluding the terminating newline character (unless there is nothing else on the line to the right of the point).
Alt-K	BACKWARD-KILL-LINE Moves to the kill history the text backward from the point to the beginning of the current line.
Ctrl-W	KILL-REGION Moves to the kill history the text between the current mark and the point.
Alt-W	SAVE-REGION Copies the text between the current mark and the point to the kill history, without deleting the text from the buffer.
Ctrl-Z Y	DISPLAY-KILL-HISTORY Displays in a type-out window all entries contained in the kill history. An arrow marks the current "top entry".
Ctrl-Z O	APPEND-NEXT-KILL Causes the next kill command to either append or prepend the killed text to the entry at the top of the kill history. A "backward kill" prepends, and a "forward kill" appends, when the killed object is a word or a line. Similarly for KILL-REGION and SAVE-REGION (where "backward" means a region backward from the point to the mark, and "forward" means a region forward from the point to the mark).

If you give a sequence of kill commands without having given any intervening commands except cursor-motion commands, then the texts being killed are contiguous in the edit buffer. They will be *automatically* strung together in the kill history also (appended or prepended to the first-killed text). The single entry in the kill history which is thus built up will therefore be a copy of the entire block of text in the buffer that was killed by the sequence of commands.

The *kill history* is a push-down list somewhat like the mark pdl; but there are important differences. Each entry is a piece of text; and each entry was put on the list by a kill command. A new entry pushes down the existing entries. However, there is a maximum of five entries; if there are five entries and a new entry is made, then the fifth -- the oldest entry -- is lost.

The YANK and YANK-POP commands recover entries from the kill history. The overall effect of YANK or YANK-POP is to copy a text entry from the kill history to the current point in the edit buffer. Neither command changes either the contents or the order of the entries in the kill history.

The general idea of using these commands is that you use YANK to recover the top entry from the kill history; and you use a series of YANK-POP commands to recover a lower-down entry. In detail, YANK and YANK-POP operate as follows:

Ctrl-Y YANK
This command copies the top text entry from the kill history to the point (in the edit buffer).

Alt-Y YANK-POP
There are three different cases, depending on what preceded the YANK-POP command:

- **The preceding command was not YANK or YANK-POP.** Then YANK-POP has the same effect as YANK.

- **The preceding command was YANK.** Then YANK-POP copies the second entry in the kill history to the edit buffer, replacing the text of the first entry which was copied to the edit buffer by YANK.

- **The preceding command was YANK-POP.** Then YANK-POP copies the next-lower entry in the kill history to the edit buffer, replacing the text of the preceding entry which was copied to the edit buffer by the preceding YANK-POP command. That is: if a YANK-POP command had copied the second entry, then another immediate YANK-POP command would copy the third entry. If the preceding entry is the lowest entry in the kill history, then YANK-POP copies the highest entry.

The net effect of the kill commands and the YANK and YANK-POP commands is to enable you to delete, move and copy any block of text at all by first moving it to the kill history with a kill command, and then recovering it, if wanted, to the same location or a new one with a YANK or YANK-POP command.

The following example illustrates killing and recovering texts. Two lines (marked L1 and L2) in an edit buffer are deleted one after the other, and then returned to the buffer by YANK and YANK-POP. Note that in this series of diagrams the cursor is moved only once, between the two executions of the KILL-LINE command (i.e., between the second and fourth frames). Thus the three last commands -- YANK and two YANK-POP commands -- are given without moving the cursor.

```
     EDIT BUFFER              KILL HISTORY

 _______________________     ________________
| L1                    |   |                |
| L2                    |   |________________|
|                       |   |                |
|_______________________|   |________________|

              KILL-LINE [Ctrl-K]
 _______________________     ________________
|                       |---->|      L1        |
| L2                    |   |________________|
|                       |   |                |
|_______________________|   |________________|

              NEXT-LINE [Ctrl-N]
 _______________________     ________________
|                       |---->|      L1        |
| L2                    |   |________________|
|                       |   |                |
|_______________________|   |________________|

              KILL-LINE [Ctrl-K]
 _______________________     ________________
|                       |---->|      L2        |
| _                     |   |________________|
|                       |   |      L1        |
|_______________________|   |________________|

                YANK [Ctrl-Y]
 _______________________     ________________
|                       |<----|      L2        |
| L2_                   |   |________________|
|                       |   |      L1        |
|_______________________|   |________________|

              YANK-POP [Alt-Y]
 _______________________     ________________
|                       |   |      L2        |
| L1_                   |   |________________|
|                       |<----|      L1        |
|_______________________|   |________________|

              YANK-POP [Alt-Y]
 _______________________     ________________
|                       |<----|      L2        |
| L2_                   |   |________________|
|                       |   |      L1        |
|_______________________|   |________________|
```

2.3.11 Editing in Two Windows

You can split the edit-window area on the edit screen into two edit windows. All of the editing commands will apply to only one window at a time. Then it is easy and fast to edit almost simultaneously in the two windows.

Each window has an edit buffer associated with it. The two buffers may be the same buffer; or they may be different buffers, enabling you to edit two different files.

At any particular time, the cursor will be in one of the windows, called the *current window*. Any input that you type applies to the current window and the current point.

To work in the other window, give the OTHER-WINDOW command (**Ctrl-X O**). This makes the other window the current window. GMACS maintains any needed information about the inactive window so that when you return there, you can pick up where you left off. In particular, the point is maintained. There is also a mark pdl (see section 2.3.9) for each buffer; so, there are two mark pdl's unless the two windows have the same buffer. However, GMACS maintains only one kill history, which is accessible in both windows. This feature is one of the main reasons for editing in two windows: it enables you to merge text between two buffers with minimum effort.

Here are the commands for two-window manipulation:

Ctrl-X 2 TWO-WINDOWS
Splits the edit window into two windows, with the upper window showing the buffer which was in the single window, and the lower window showing the previously-edited buffer, if any. (If there is none, the two windows show the same buffer.) The upper window becomes the current window.

Ctrl-Z V SCROLL-OTHER-WINDOW
Scrolls the other window forward by one screen.

Ctrl-X O OTHER-WINDOW
Moves the cursor to the other window, which becomes the current window.

Ctrl-X 1 ONE-WINDOW
Returns the screen to single window display. If no prefix is used, the current window becomes the single window; with the prefix **Ctrl-U**, the other window becomes the single window.

2.4 Editing LISP

This section describes those GMACS commands which are designed specifically to manipulate LISP language constructs. The language constructs which can be edited by these commands are the basic ones in LISP: symbols and lists and other s-expressions. The facility for these manipulations, and for evaluating LISP code directly from within GMACS (also described in this section), constitutes a significant interactive program-development tool.

Since LISP code is written as lines of text, all of the GMACS commands already described in this chapter can of course also be applied to lines of LISP code. However, the special feature of the commands in this section is that they apply to lists and other s-expressions as the basic objects of manipulation, rather than to words or lines.

Several of the commands refer to "the end of the current list", or "the beginning of the current s-expression", or similar points. For this to make sense, it's necessary to know what the "current" item means for an s-expression or a function definition or a list: The current item is the lowest-level item of that kind containing the point. The "next" item is the first item of that kind encountered, in one search direction or the other (the search direction is always specified).

The "beginning" and "end" of an item need to be defined also. Beginning and end are marked in LISP code by delimiting characters; for the items of interest, these are as follows:

- For an atom: parentheses or white space (the space, tab, or newline character)

- For a list: parentheses

If a command specifies an action on a current or a previous or a next item, and there is no such item in the edit buffer, then GMACS rings the bell and does not move the point. (In other words, the command has no effect in that instance except to ring the bell.)

2.4.1 Cursor Motion

These commands move the cursor to the beginning or the end of the current s-expression.

Ctrl-Z B BACKWARD-SEXP
If the preceding character is not (,), or white space, the point is moved to just left of the first character of the current s-expression.

If the preceding character is), the point is moved to just left of the matching (.

If the preceding character is white space, the point is moved to just left of the first character of the preceding s-expression.

If the preceding character is (, the point moves to the left of it.

Ctrl-Z F FORWARD-SEXP
If the next character is not (,), or white space, the point is moved to just right of the last character of the current s-expression.

If the next character is (, the point is moved to just right of the matching).

If the next character is white space, the point is moved to just right of the last character of the next s-expression.

If the next character is), the point moves to the right of it.

These commands move the cursor to the beginning or the end of the current list.

Ctrl-Z P BACKWARD-LIST
Searches backward, positioning the point just before the first open parenthesis encountered at the same level.

Ctrl-Z N FORWARD-LIST
Searches forward, positioning the point just after the first close parenthesis encountered at the same level.

The command DOWN-LIST enables you to move the point into a list nested within the current list.

Ctrl-Z D DOWN-LIST
Searches forward, positioning the point just after the next open parenthesis within the current list. Beeps and does not move the point if a close parenthesis is encountered first.

These two sample screens illustrate the effect of DOWN-LIST:

(before DOWN-LIST command)

```
|
|  (+ a (+ b (+ c  d)))
|
|___________________________________
```

(after DOWN-LIST command)

```
|
|  (+ a (+ b (+ c d)))
|
|___________________________________
```

Note that DOWN-LIST is a forward move. There is no "backward-down-list" command.

These two commands enable you to move the cursor from the current nested list to the list which contains it:

Ctrl-Z (BACKWARD-UP-LIST
Searches backward, positioning the point just before the first unmatched open parenthesis.

Ctrl-Z) FORWARD-UP-LIST
Searches forward, positioning the point just after the first unmatched close parenthesis.

If the point is not currently within a list, then the terminal beeps and the point is not moved.

These two sample screens show the effect of the FORWARD-UP-LIST command:

(before FORWARD-UP-LIST command)

```
|
|   (+ a (+ b (+ c d)))
|
|___________________________________
```

(after FORWARD-UP-LIST command)

```
|
|   (+ a (+ b (+ c d)))_
|
|___________________________________
```

These two commands enable you to move the point to the beginning or to the end of the current function definition. (It's assumed that a function definition (and any other form which is not nested within another form) always begins in column 1 of a line.)

Ctrl-Z A BEGINNING-OF-DEFINITION
Searches backward, positioning the point just before the first open parenthesis encountered in column 1 of a line.

Ctrl-Z E END-OF-DEFINITION
If the point is currently in a function definition, performs a BEGINNING-OF-DEFINITION and then a FORWARD-SEXP, leaving the point just after the close parenthesis matching the definition's first open parenthesis. If the point is not in a current definition, the point is moved to the end of the next definition.

2.4.2 Convenience Aids to Writing in LISP

Three miscellaneous GMACS features aid you in writing LISP programs. They are the MAKE-EMPTY-LIST command; and the *paren-flash* and *paren-beep* features (which are not commands).

Alt-9 MAKE-EMPTY-LIST
Inserts matching parentheses around the point.

Whenever the point is just to the right of a close parenthesis, the corresponding open parenthesis blinks on the screen (if it appears in the window). This is the *paren-flash* feature. It is enabled automatically in GMACS. To disable it, give the GCLISP command **(setf *flash-mode* nil)** after starting up GMACS. (That is, leave GMACS, give the command, and re-enter GMACS. Another way to disable the feature is to put this command into the GMACS initialization file GMINIT.LSP.) To re-enable paren-flash, give the GCLISP command **(setf *flash-mode* t)**.

Whenever a close parenthesis is typed, your terminal will beep, and the message **No matching open parenthesis** will be printed, if there is no matching open parenthesis anywhere in the buffer. (The matching open parenthesis need not be visible in the window.) This is the *paren-beep* feature.

2.4.3 Indenting LISP Expressions

These commands enable you to indent a line of LISP code to reflect the nesting level of the current form.

Ctrl-Z Q INDENT-SEXP
Corrects the indentation of the s-expression to the right of the point.

Ctrl-I INDENT-TO-LEVEL
Indents the current line to the appropriate level with respect to the preceding line, moving the code on the line to the right or left as needed. The position of the point is left unchanged in relation to the text.

Ctrl-J or Ctrl-ENTER
INDENT-NEWLINE
Inserts a newline character at the point and then performs an INDENT-TO-LEVEL on the new line thus created.

Alt-3 INDENT-FOR-COMMENT
If the current line has no comment, moves the point out to the comment column (inserting spaces as necessary) and inserts a semi-colon. If the line already has a comment, the comment is indented the correct number of spaces and the point is positioned to the right of the semi-colon.

2.4.4 Displaying Information About LISP Code

Several commands enable you to display on-line documentation about LISP functions. The documentation comes from the text which would be displayed in response to the GMACS help command ED-DOC (invoked by **Alt-H D**).

Ctrl-Z L DISPLAY-LAMBDA-LIST
Displays in the echo area the lambda list of the current function (the function at the beginning of the current s-expression).

Ctrl-Z ? DISPLAY-DOCUMENTATION
Displays in a type-out window the full Help documentation of the current function.

Alt-2 DISPLAY-MACROEXPANSION
Displays in a type-out window the macro-expansion of the s-expression immediately to the right of the point.

2.4.5 Killing and Recovering LISP Code

These commands enable you to kill s-expressions and comments. As described earlier, "killing" text means removing it from the edit buffer and moving it to the kill history. Like any entry in the kill history, it can then be recovered by YANK and YANK-POP commands for insertion, if desired, elsewhere in the buffer or in another buffer. See section 2.3.10, "Killing and Recovering Text".

Ctrl-Z K KILL-SEXP
Moves to the kill history the characters from the point forward through the end of the s-expression immediately to the right. The command has the same effect as the command sequence SET-POP-MARK (**Ctrl-@**), then FORWARD-SEXP, and then KILL-REGION.

Ctrl-Z Rubout BACKWARD-KILL-SEXP
Moves to the kill history the characters from the point backward to the beginning of the s-expression immediately to the left.

Ctrl-Z ; KILL-COMMENT
Moves to the kill history any comment on the current line (that is, all of the characters from the first semi-colon through the last character before the newline).

2.4.6 Evaluating LISP Code from the Editor

Without leaving the GMACS environment, you can call on GCLISP to evaluate LISP code you are editing and print the results to the screen. The effect is virtually the same as if the code were being loaded from an existing file in the interpreter environment. This facility saves you the time and trouble of writing out the code from the edit buffer to an on-line file, leaving GMACS, loading the file, and returning to GMACS. The result is much faster program editing and debugging.

The evaluation results are printed to a type-out window. If there is an error in the code, or if you type **Ctrl-Break** during the evaluation (or if the **break** function is part of the code), the evaluation behavior is the same as if you were typing the code form-by-form interactively. Evaluation and printing of results are suspended; a new level of the listener is invoked; and you can then perform debugging operations: viewing the current values of variables, tracing the execution stack, and so forth. You continue via **Ctrl-G** (from an error) or **Ctrl-P** (from a break), as always in the listener.

Whether there was an error or not, GCLISP returns to the GMACS environment only when evaluation and printing are complete. You can then pick up editing where you left off -- in particular, revising the forms where errors were found.

These are the commands which invoke evaluation.

Alt-1 EVAL-SEXP
Invokes evaluation of the s-expression to the right of the point. The point is not moved.

Ctrl-Z C EVAL-DEFINITION
Evaluates the current function (the function which would be found by the command BEGINNING-OF-DEFINITION). The point is not moved.

2.5 Table of Function Keys

The following table lists the IBM PC keyboard function keys and the GMACS commands which they invoke.

KEY	COMMAND NAME
F1	EXIT-EDITOR
F2	ED-HELP
F3	SELECT-BUFFER
F4	SELECT-PREVIOUS-BUFFER
F5	LIST-BUFFERS
F6	DISPLAY-DIRECTORY
F7	FIND-FILE
F8	READ-FILE
F9	SAVE-FILE
F10	WRITE-FILE

2.6 Table of Cursor Motion Keys

This table shows the GMACS commands invoked by the IBM PC keyboard cursor motion keys and the **Insert** and **Delete** keys. All of these keys are bound to GMACS commands; and six of them also invoke GMACS commands in a keychord with the **Ctrl** key.

KEY	COMMAND NAME
Home	BEGINNING-OF-BUFFER
Up Arrow	PREVIOUS-LINE
Pg Up	PREVIOUS-SCREEN
Left Arrow	BACKWARD-CHAR
Right Arrow	FORWARD-CHAR
End	END-OF-BUFFER
Down Arrow	NEXT-LINE
Pg Dn	NEXT-SCREEN
Ins	OPEN-LINE
Del	DELETE-CHAR
Ctrl-Left Arrow	BACKWARD-WORD
Ctrl-Right Arrow	FORWARD-WORD
Ctrl-Pg Up	BACKWARD-SEXP
Ctrl-Pg Dn	FORWARD-SEXP
Ctrl-Home	BEGINNING-OF-DEFINITION
Ctrl-End	END-OF-DEFINITION

2.7 Summary GMACS Command Reference (by Topic)

This section provides a summary listing of GMACS editor commands, with their key bindings and meanings. The commands are grouped by topic (e.g., search and replace commands).

2.7.1 Cursor Motion Commands

KEY	**COMMAND NAME AND FUNCTION**
Ctrl-F or Right Arrow	FORWARD-CHAR Moves the point one character position to the right (forward).
Ctrl-B or Left Arrow	BACKWARD-CHAR Moves the point to the left (back) one character position.
Alt-F or Ctrl-Right Arrow	FORWARD-WORD Moves the point forward to the end of the current word.
Alt-B or Ctrl-Left Arrow	BACKWARD-WORD Moves the point backward to the beginning of the current word.
Ctrl-A	BEGINNING-OF-LINE Moves the point to the beginning of the current line.
Ctrl-E	END-OF-LINE Moves the point to the end of the current line.
Ctrl-N or Down Arrow	NEXT-LINE Moves the point forward to the same column in the next line.
Ctrl-P or Up Arrow	PREVIOUS-LINE Moves the point backward to the same column in the preceding line.

Ctrl-Z < or Home
BEGINNING-OF-BUFFER
Positions the point before the first character in the edit buffer.

Ctrl-Z > or End
END-OF-BUFFER
Positions the point after the last character in the edit buffer.

2.7.2 Edit Window Commands

KEY	**COMMAND NAME AND FUNCTION**
Ctrl-V or PgDn	NEXT-SCREEN Moves the window forward in the edit buffer by about one window-length (one edit screen). The window is positioned on the edit buffer so that the previous last line in the window becomes the new first line.
Alt-V or PgUp	PREVIOUS-SCREEN Moves the window backward in the edit buffer by about one window-length (one edit screen). The window is positioned on the edit buffer so that the previous first line in the window becomes the new last line.
Ctrl-X 2	TWO-WINDOWS Splits the edit window display area into two windows, with the upper window showing the current buffer and the lower window showing the previous buffer. The upper window becomes the current window.
Ctrl-X O	OTHER-WINDOW Moves the cursor to the other window, which becomes the current window.
Ctrl-Z V	SCROLL-OTHER-WINDOW Scrolls the other window forward one screen.
Ctrl-X 1	ONE-WINDOW Returns the editor display to one window by expanding the current window to the size of the terminal display.
Ctrl-L	REDISPLAY-SCREEN Completely redisplays the screen, leaving the point near the middle of the edit window.

2.7.3 Text Deletion Commands

KEY	COMMAND NAME AND FUNCTION
Ctrl-D or Del	DELETE-CHAR Deletes the character to the right of the point.
Ctrl-H or Rubout	RUBOUT Deletes the character to the left of the point.
Ctrl-^	DELETE-INDENTATION Deletes the newline character and any indentation at the beginning of the current line. This action appends the current line to the preceding line.
Ctrl-\	DELETE-HORIZONTAL-SPACE Deletes any spaces or tabs adjoining the point on either side.
Alt-D	KILL-WORD Moves the word to the right of the point to the kill history.
Ctrl-Rubout	BACKWARD-KILL-WORD Moves the word to the left of the point to the kill history.
Ctrl-K	KILL-LINE Moves all characters to the right of the point on the current line to the kill history, not including the terminating Newline character. (If Newline is the only character to the right of the point on the current line, it is moved to the kill history.)
Alt-K	BACKWARD-KILL-LINE Moves all characters to the left of the point on the current line to the kill history.
Ctrl-W	KILL-REGION Moves the characters between the current mark and the point to the kill history.

2.7.4 Buffer and File Commands

KEY	**COMMAND NAME AND FUNCTION**
Ctrl-X Ctrl-F or F7	FIND-FILE Searches the set of edit-buffer names for a specified filename. Selects the buffer with that filename if there is one. Otherwise, creates a buffer with that name and reads the file into the new buffer from disk. The command prompts you for the filename.
Ctrl-X Ctrl-R or F8	READ-FILE Reads a specified file into the current buffer, overwriting the existing contents of the buffer. The command prompts for the filename.
Ctrl-X Ctrl-S or F9	SAVE-FILE Copies the contents of the current edit buffer into disk storage under the current file name. If a file with that name already exists on disk, the command copies over the existing file.
Ctrl-X B or F3	SELECT-BUFFER Selects a specified buffer and displays it in the edit window. The command prompts you for the name of the desired buffer. Pressing the ENTER key without entering a buffer name selects the previous buffer. If the buffer does not exist, a new buffer is opened having no current file.
Ctrl-X K	KILL-BUFFER Prompts for the name of a buffer and removes it from the list of buffers known to the editor.
Ctrl-X P or F4	SELECT-PREVIOUS-BUFFER Selects the previous buffer.
Ctrl-X Ctrl-B or F5	LIST-BUFFERS Lists the names of all existing buffers in a type-out window, together with the name of associated files, if any. Modified buffers

	are marked with the buffer-status (*).
Ctrl-X U	UNMODIFY-BUFFER Marks the current buffer as unmodified since it was last read from a file or written to a file. Clears the buffer-status (*) in the mode line.
Ctrl-X Ctrl-W or F10	WRITE-FILE Writes out the contents of the current buffer to the specified file. The command prompts you for the filename.
Ctrl-X C	CHANGE-DIRECTORY Prompts for a directory name, and changes the current default directory to the directory with that name.
Ctrl-X Ctrl-D or F6	DISPLAY-DIRECTORY Prompts for a pathname and displays a list of all files that match it.

2.7.5 Search and Replace Commands

KEY	**COMMAND NAME AND FUNCTION**
Ctrl-S	FORWARD-SEARCH Searches forward from the point for a specified character string. The point moves to the end of the first instance found. The command prompts you for the string.
Ctrl-R	REVERSE-SEARCH Searches backward from the point for a specified character string. The point moves to the beginning of the first instance found. The command prompts for the string.
Alt-%	QUERY-REPLACE Replaces selected instances of a character string from the point to the end of the buffer, with another specified string. At each occurrence, you are queried as to whether or not to replace it. The command prompts for both strings.
Alt-*	GLOBAL-REPLACE Replaces all instances of a specified string with another string, from the point to the end of the buffer. The command prompts for both strings.

2.7.6 Case-Setting Commands

KEY	COMMAND NAME AND FUNCTION
Alt-C	UPPERCASE-INITIAL Capitalizes the first letter of the word to the right of the point and puts the other characters in lowercase.
Alt-L	LOWERCASE-WORD Puts the word to the right of the point in lowercase.
Alt-U	UPPERCASE-WORD Puts the word to the right of the point in uppercase.
Ctrl-X Ctrl-U	UPPERCASE-REGION Puts all the letters in the region in uppercase.
Ctrl-X Ctrl-L	LOWERCASE-REGION Puts all the letters in the region in lowercase.

2.7.7 Commands for Editing LISP

KEY	COMMAND NAME AND FUNCTION
Ctrl-Z K	KILL-SEXP Moves to the kill history the characters forward from the point through the end of the current s-expression.
Ctrl-Z Rubout	BACKWARD-KILL-SEXP Moves to the kill history the characters backward from the point to the beginning of the current s-expression.
Ctrl-Z ;	KILL-COMMENT Moves to the kill history any comment on the current line (that is, all of the characters from the first semi-colon through the last character before the newline).
Ctrl-Z F or Ctrl-PgDn	FORWARD-SEXP Moves the point to the end of the s-expression to its right.

Ctrl-Z B or Ctrl-PgUp
BACKWARD-SEXP
Moves the point to the beginning of the s-expression to its left.

Ctrl-Z N
FORWARD-LIST
Moves the point to the end of the list to its right. The command searches for a close parenthesis and positions the point just after it.

Ctrl-Z P
BACKWARD-LIST
Moves the point to the beginning of the list to its left. The command searches for an open parenthesis and positions the point just to the left of it.

Ctrl-Z D
DOWN-LIST
Moves the point forward in the edit buffer until it is just to the right of the next open parenthesis.

Ctrl-Z U, Ctrl-Z (
BACKWARD-UP-LIST
Searches backward for an unmatched open parenthesis and positions the point to the left of the first one encountered.

Ctrl-Z)
FORWARD-UP-LIST
Searches forward for an unmatched close parenthesis and positions the point to the right of the first one encountered.

Ctrl-Z A, Ctrl-Z [, Ctrl-Home
BEGINNING-OF-DEFINITION
Moves the point backward to the beginning of the current LISP function.

Ctrl-Z E, Ctrl-Z], Ctrl-End
END-OF-DEFINITION
Moves the point forward to the end of the current LISP function.

Alt-!
EVAL-SEXP
Evaluates the s-expression to the right of the point.

Ctrl-Z C
EVAL-DEFINITION
Evaluates the current function.

Ctrl-Z Q
INDENT-SEXP
Corrects the indentation of the s-expression to the right of the point.

Ctrl-I INDENT-TO-LEVEL
Indents the current line correctly.

Alt-3 INDENT-FOR-COMMENT
If the current line has no comment, moves the point out to the comment column (inserting spaces as necessary) and inserts a semi-colon. If the line already has a comment, the comment is indented the correct number of spaces and the point is positioned to the right of the semi-colon.

Ctrl-J or Ctrl-Enter
INDENT-NEWLINE
Inserts a newline character at the current point, moves the point to the new line, and inserts white space to correctly indent the new line. The point is placed to the right of the indentation.

Alt-9 MAKE-EMPTY-LIST
Inserts matching parentheses around the point.

Ctrl-Z L DISPLAY-LAMBDA-LIST
Displays in the echo window the lambda-list of the current function definition.

Ctrl-Z ? DISPLAY-DOCUMENTATION
Displays in a type-out window the full Help documentation of the current function definition.

Alt-2 DISPLAY-MACROEXPANSION
Displays in a type-out window the macro-expansion of the current s-expression.

2.7.8 Region and Kill History Commands

KEY **COMMAND NAME AND FUNCTION**

Ctrl-X Ctrl-X or Ctrl-Z Space
EXCHANGE-POINT-AND-MARK
Exchanges the point and the current mark.

Ctrl-@ SET-POP-MARK
Puts a mark where the point is and puts it at the top of the mark pdl (making it the current mark). Prefixed with **Ctrl-U**, the command positions the point at the current mark and pops that mark from the pdl. Prefixed with **Ctrl-U Ctrl-U**, the command just pops the current mark from the mark pdl.

Alt-W	SAVE-REGION Moves a copy of a region to the kill history without erasing it from the edit buffer.
Ctrl-Y	YANK Inserts the entry at the top of the kill history into the current buffer at the point.
Alt-Y	YANK-POP If the last command was YANK or YANK-POP, the text returned to the buffer by the last command is replaced in the buffer by the next lower entry in the kill history. Otherwise the command has the same effect as YANK.
Ctrl-Z O	APPEND-NEXT-KILL Causes the next kill command to append the killed text to the entry at the top of the kill history.
Ctrl-Z Y	DISPLAY-KILL-HISTORY Displays in a type-out window all entries contained in the kill history.

2.7.9 Miscellaneous Commands

KEY	**COMMAND NAME AND FUNCTION**
Ctrl-X Ctrl-C or F1	EXIT-EDITOR Exits the GMACS environment and returns you to the GCLISP environment from which you entered GMACS.
Ctrl-Break	(Break to listener)
Esc	DEADEND Aborts the current command and returns you to normal GMACS command entry.
Ctrl-G, Ctrl-X Ctrl-G, F2 G, Alt-H G	ED-BEEP Aborts the current command, rings the terminal bell, and returns you to normal GMACS command entry.
Alt-H or F2	HELP-DEADEND Displays a help menu that lists the options for accessing information relating to GMACS commands and key bindings.

Alt-H ?, Alt-H H, F2 ?, F2 H
ED-HELP
Displays the help guide consisting of descriptions of the options listed in the help menu.

Alt-H A or F2 A
ED-APROPOS
Prompts you for a character string, and displays in a type-out window every GMACS command which contains the specified string in its name.

Alt-H K or F2 K
ED-KEYCHORD
Prompts you for a keychord, and displays in a type-out window the command associated with the specified keychord.

Alt-H D or F2 D
ED-DOC
Prompts you for a character string, and displays in a type-out window the on-line documentation for every GMACS command which contains the specified string in its name.

Alt-H T or F2 T
ED-TEACH
Invokes the GMACS on-line tutorial.

Alt-X
EXTENDED-COMMAND
Any LISP function not requiring an argument, and any GMACS command, including those GMACS commands not bound to a keychord or key sequence, can be invoked by entering **Ctrl-X** and typing the name of the command.

Enter or Ctrl-M
NEWLINE
Inserts a newline character at the point. Any characters to the right of the point move to the new line. The point is moved to the first position of the new line.

Ctrl-O or Ins
OPEN-LINE
Inserts a newline character after the point (unlike <ENTER>, which inserts the newline before the point).

Ctrl-U
NUMERIC-ARG
Used as a command prefix to establish a repeat count for the command (valid for most commands). Prefixed by **Ctrl-U**, a command

executes 4 times (the default repeat count is 4). Prefixed by **Ctrl-U <n>**, a command executes **<n>** times. If **<n>** is negative and there is a meaningful "opposite" version of the command, that is executed positive-**<n>** times. (For example, the command to move the cursor down by -4 lines will move the cursor up by 4 lines.) Repetitions of **Ctrl-U** following the numeric argument **<n>**, if any, multiply the repeat count by 4 each time.

Ctrl-Q QUOTED-INSERT
Used for inserting as text those characters which otherwise act as editing commands. The character typed after **Ctrl-Q** is inserted into the buffer.

Ctrl-T EXCHANGE-CHARACTERS
Transposes the two characters to the left of the point.

2.8 GMACS Commands: Quick-Reference Table

This section lists the key bindings and command names of GMACS editor commands for quick referencing.

2.8.1 Cursor Motion Commands

KEY	COMMAND NAME
Ctrl-F or Right Arrow	FORWARD-CHAR
Ctrl-B or Left Arrow	BACKWARD-CHAR
Alt-F or Ctrl-Right Arrow	FORWARD-WORD
Alt-B or Ctrl-Left Arrow	BACKWARD-WORD
Ctrl-A	BEGINNING-OF-LINE
Ctrl-E	END-OF-LINE
Ctrl-N or Down Arrow	NEXT-LINE
Ctrl-P or Up Arrow	PREVIOUS-LINE
Ctrl-Z < or Home	BEGINNING-OF-BUFFER
Ctrl-Z > or End	END-OF-BUFFER

2.8.2 Edit Window Commands

KEY	COMMAND NAME
Ctrl-V or PgDn	NEXT-SCREEN
Alt-V or PgUp	PREVIOUS SCREEN

Ctrl-X 2	TWO-WINDOWS
Ctrl-X O	OTHER-WINDOW
Ctrl-Z V	SCROLL-OTHER-WINDOW
Ctrl-X 1	ONE-WINDOW
Ctrl-L	REDISPLAY-SCREEN

2.8.3 Text Deletion Commands

KEY	COMMAND NAME
Ctrl-D or Del	DELETE-CHAR
Ctrl-H or Rubout	RUBOUT
Ctrl-^	DELETE-INDENTATION
Ctrl-	DELETE-HORIZONTAL-SPACE
Alt-D	KILL-WORD
Ctrl-Rubout	BACKWARD-KILL-WORD
Ctrl-K	KILL-LINE
Alt-K	BACKWARD-KILL-LINE
Ctrl-W	KILL-REGION

2.8.4 Buffer and File Commands

KEY	COMMAND NAME
Ctrl-X Ctrl-F or F7	FIND-FILE
Ctrl-X Ctrl-R or F8	READ-FILE
Ctrl-X Ctrl-S or F9	SAVE-FILE
Ctrl-X B or F3	SELECT-BUFFER
Ctrl-X K	KILL-BUFFER

Ctrl-X P or F4	SELECT-PREVIOUS-BUFFER
Ctrl-X Ctrl-B	LIST-BUFFERS
Ctrl-X U	UNMODIFY-BUFFER
Ctrl-X Ctrl-W or F10	WRITE-FILE
Ctrl-X C	CHANGE-DIRECTORY
Ctrl-X Ctrl-D	DISPLAY-DIRECTORY

2.8.5 Search and Replace Commands

KEY	COMMAND NAME
Ctrl-S	FORWARD-SEARCH
Ctrl-R	REVERSE-SEARCH
Alt-%	QUERY-REPLACE
Alt-*	GLOBAL-REPLACE

2.8.6 Case-Setting Commands

KEY	COMMAND NAME
Alt-C	UPPERCASE-INITIAL
Alt-L	LOWERCASE-WORD
Alt-U	UPPERCASE-WORD
Ctrl-X Ctrl-U	UPPERCASE-REGION
Ctrl-X Ctrl-L	LOWERCASE-REGION

2.8.7 Commands for Editing LISP

KEY	COMMAND NAME
Ctrl-Z K	KILL-SEXP
Ctrl-Z Rubout	BACKWARD-KILL-SEXP
Ctrl-Z ;	KILL-COMMENT

Ctrl-Z F or Ctrl-PgDn
FORWARD-SEXP

Ctrl-Z B or Ctrl-PgUp
BACKWARD-SEXP

Ctrl-Z N	FORWARD-LIST
Ctrl-Z P	BACKWARD-LIST
Ctrl-Z D	DOWN-LIST

Ctrl-Z U or Ctrl-Z (
BACKWARD-UP-LIST

Ctrl-Z)	FORWARD-UP-LIST

Ctrl-Z A, Ctrl-Z [, or Ctrl-HOME
BEGINNING-OF-DEFINITION

Ctrl-Z E, Ctrl-Z], or Ctrl-END
END-OF-DEFINITION

Alt-1	EVAL-SEXP
Ctrl-Z C	EVAL-DEFINITION
Ctrl-Z Q	INDENT-SEXP
Ctrl-I	INDENT-TO-LEVEL
Alt-3	INDENT-FOR-COMMENT

Ctrl-J or Ctrl-Enter
INDENT-NEWLINE

Alt-9	MAKE-EMPTY-LIST
Ctrl-Z L	DISPLAY-LAMBDA-LIST
Ctrl-Z ?	DISPLAY-DOCUMENTATION
Alt-2	DISPLAY-MACROEXPANSION

2.8.8 Region and Kill History Commands

KEY	**COMMAND NAME**

Ctrl-X Ctrl-X or Ctrl-Z Space
EXCHANGE-POINT-AND-MARK

Ctrl-@	SET-POP-MARK

Alt-W	SAVE-REGION
Ctrl-Y	YANK
Alt-Y	YANK-POP
Ctrl-Z O	APPEND-NEXT-KILL
Ctrl-Z Y	DISPLAY-KILL-HISTORY

2.8.9 Miscellaneous Commands

KEY	**COMMAND NAME**
Ctrl-X Ctrl-C or F1	EXIT-EDITOR
Ctrl-Break	(Break to listener)
Esc	DEADEND
Ctrl-G or Ctrl-X Ctrl-G or Alt-H G or F2 G	ED-BEEP
Alt-H or F2	HELP-DEADEND
Alt-H ? OR F2 ?	ED-HELP
Alt-H A or F2 A	ED-APROPOS
Alt-H K or F2 K	ED-KEYCHORD
Alt-H D or F2 D	ED-DOC
Alt-H T or F2 T	ED-TEACH
Alt-X	EXTENDED-COMMAND
Enter or Ctrl-M	NEWLINE
Ctrl-O or Ins	OPEN-LINE
Ctrl-U	NUMERIC-ARG
Ctrl-Q	QUOTED-INSERT

Ctrl-T EXCHANGE-CHARACTERS

Chapter 3

On-Line Help Facilities

This chapter describes the main on-line help facilities of GCLISP. These facilities are there to aid you when you are constructing a GCLISP program and need information about particular functions or symbols.

When you press **Alt-H**, the resulting display shows the types of help available and how to invoke them:

```
 ____________________________________________________________

 To invoke one of the following GCLISP applications,
 type the indicated keychord:

   Alt-E    The LISP Explorer, an on-line tutorial
  Ctrl-E    The GMACS Editor

 To get help in one of the following areas,
 type the indicated keychord:

   Alt-K    "Keys" - Displays a list of the actions
            invoked by special keys and keychords.

   Alt-A    "Apropos"  - Lists all symbols whose names
            contain a specified string. Prompts for the string.

   Alt-D    "Documentation" - Displays the on-line documentation
            for a specified function, variable, or type name.
            Prompts for the name.

   Alt-L    "Lambda-List" - Displays the arguments for a
            specified function.  Prompts for the function name.
 ____________________________________________________________
* _
```

Alt-A, **Alt-D**, and **Alt-L** give detailed information about GCLISP functions and symbols. These specific help options correspond to GCLISP functions:

- "Apropos": the **apropos** function

- "Documentation": the **doc** function

- "Lambda List": the **lambda-list** function

That is, you can get each kind of help either by typing the keychord -- for example, **Alt-A** -- or by a function call -- for example, **(apropos** *string***)**. The sections of this chapter describe each of these options under its function name. (No GCLISP function corresponds to **Alt-K**, the "Keys" help (described in section 1.4, "Keychord Commands to the Interpreter").)

3.1 APROPOS

apropos prints to the screen the names of all LISP symbols that contain the string specified as the **apropos** argument. This function is particularly useful for looking up LISP symbols with names you cannot remember.

The type of each LISP symbol (e.g., "function") is also printed.

There are no restrictions on the string argument. (In particular, you can give the null string as the argument; then the names of all currently-defined LISP symbols will be printed to the screen, because the null string is contained in every name. This has the same effect as typing **Alt-H A <ENTER>**.)

By way of illustration, suppose that we give the symbol **foo** the following function definition:

```
* (defun foo (a b) (+ a b))
FOO
* _
```

If we now apply **apropos** to the symbol **foo**, this screen appears:

```
* (apropos 'foo)
FOO - function, arglist: (A B)

NIL
* _
```

The response shows that **foo** is currently the only LISP symbol whose print-name contains the sequence of letters "FOO". Furthermore, the symbol **foo** is a function name; and its *arglist*[9] is **(A B)**.

----

9. Another name for *lambda-list*. See section 3.3, "LAMBDA-LIST".

The function **apropos** returns the value **nil**, as shown above. (The LISP names that **apropos** prints to the screen are not returned values.)

With **foo** already defined as a function, we can further define **foo** as a variable and assign it the string "foo adds two numbers" as follows:

(setf foo "foo adds two numbers")

If **apropos** is now applied to the string "foo", the response is different from before:

```
* (apropos 'foo)
FOO - bound
FOO - function, arglist: (A B)

NIL
* _
```

The new entry for **foo** in this display indicates that **foo** is a variable bound to some value. The previous entry for the function **foo** appears as the second line in the display.

If the string argument in the **apropos** function call is not contained in any GCLISP print-name, **apropos** simply prints the string with no information, as in this example:

```
* (apropos 'baz)
BAZ

NIL
* _
```

3.1.1 Using APROPOS to Find the Right Function

You may want to call **apropos** with the name of a particular GCLISP function as argument for the purpose of seeing what related functions are available.

For instance, you may be developing a LISP program in which a series of GCLISP objects should be put into a list. To see the names of functions, one of which might perform this task, use the string "list" as an argument to **apropos**:

```
* (apropos "list")
MULTIPLE-VALUE-LIST - special form
*PACKAGE-ALIST* - bound
LISTP - function
COPY-ALIST - function
VALUES-LIST - function
LIST - function
APROPOS-LIST - function
*LISTENER-NAME* - bound
MAKE-LIST - function
:LISTEN - bound
DOLIST - special form
IE-LAMBDA-LIST - function, arglist: (&OPTIONAL BUF IGNORE)
LIST-LENGTH - function
SETPLIST - function
SYMBOL-PLIST - function
LIST* - function
LISTENER - function
:LISTENER - bound
MAPLIST - function
LAMBDA-LIST - macro
COPY-LIST - function

NIL
* _
```

Every currently-defined GCLISP symbol that contains the string "list" in its name appears in the display (including the names of functions and variables you may have defined, as well as the names of GCLISP interpreted functions).

An arglist is included in the display produced by **apropos** only if the name names an uncompiled function. In the current example, every function entry except **ie-lambda-list** is a compiled function. (To find the arglist of a compiled function, use the **doc** function.)

The empty list -- the list () -- appears as the arglist of any function which accepts no arguments.

To find out what each of the functions listed in this example actually does, use the **doc** function, described next.

3.2 DOC

The **doc** function can help you in the way a dictionary helps you with unfamiliar words: It provides definitions of individual functions and variables.

The **doc** function call takes a LISP name as its argument, as in this example:

```
* (doc 'listp)

LISTP is a Function.
(LISTP object) -> BOOLEAN

This function is a predicate which is true
if and only if OBJECT is is of type LIST.
An object is of type LIST if and only if it
is either of type CONS or type NULL.
      (LISTP object) <=> (OR (CONSP object)
                             (NULL object))

NIL
* _
```

In this example, the first line printed to the screen says that the object named by "listp" is a function. The second line gives the syntax for the function **listp**. It says that a function call on **listp** has one argument, which can be any LISP object; and that the return value of the function call is a **boolean**.[10] A description of what the function does follows in the display. The last item printed is the **nil** return value from **doc**.

----

10. These conventions for describing LISP syntax are specified in Chapter 1 of the GCLISP Reference Manual.

The **apropos** function will display the names of both pre-defined and user-defined LISP symbols. **doc**, however, will display information only about pre-defined functions and variables, not user-defined functions and variables:

```
* (doc 'foo)
No documentation found for FOO

NIL
* _
```

3.3 LAMBDA-LIST

The function **lambda-list** is useful for finding the input requirements of a given function. **lambda-list** accepts a LISP symbol as an argument, and takes an optional second argument. If the symbol names a function, then **lambda-list** returns the function's *lambda-list*: a list of the input parameters to the function.

lambda-list behaves differently based upon the type of function named by its symbol argument:

- If the argument names an interpreted function, **lambda-list** returns the function's lambda-list and **nil**. The optional second argument is unused.

- If the argument names a compiled function and the optional argument is **nil** (or is not given), **lambda-list** searches the on-line documentation for the function, and:

    * if on-line documentation for the function is found, **lambda-list** returns the documented lambda-list and **nil**;

    * if the function's documentation is not found, **lambda-list** returns the symbols **nil** and **:not-found**.

- If the argument names a compiled function and the optional argument is present and not **nil**, **lambda-list** returns **nil** and **:not-found**.

- If the argument does not name a function, lambda-list returns **nil** and **:not-found**.

To illustrate **lambda-list** with an interpreted function, suppose that **foo** is defined as follows:

(defun foo (a b) (+ a b))

When **lambda-list** is applied to this function, the result is:

```
* (lambda-list 'foo)
(A B)
NIL
* _
```

lambda-list returned **(A B)**, the lambda-list of the function **foo.**

Here is **lambda-list** applied to a non-interpreted function (a compiled function):

```
* (lambda-list 'listp)
(|object|)
NIL
* _
```

lambda-list returned **(|object|)**, the lambda-list for **listp** as it appears in the on-line documentation.

Here is **lambda-list** applied to a symbol **baz** which does not name a function:

```
* (lambda-list 'baz)
NIL
:NOT-FOUND
* _
```

Chapter 4

Debugging in GCLISP

While building a LISP program, you may want to test it periodically to make sure that the various components function properly. If the program does not work the way you intend, you will have to find the source of the problem and correct it.

To locate problems, use the GCLISP debugging utilities. There are five of these:

- **break or Ctrl-Break**
- **backtrace or Ctrl-B**
- **trace**
- **step**
- **pprint**

In this chapter we discuss how to use these functions, individually and in combination, to debug your programs.

4.1 BREAK

The function **break** suspends the current listener and starts a new one. At this new listener, you have all of the services available at Top-Level, the level-0 listener.

break can be invoked by calling it or by depressing the keychord **Ctrl-Break**. A second type of program suspension may occur when the evaluator encounters an error. To illustrate, we use a function **foo** that takes two arguments, and apply it to only one argument. **foo** is defined as follows:

```
(defun foo (a b) (+ a b))
```

When we evaluate **foo** with only one argument, the results are as follows:

```
* (foo 2)

ERROR:
Not enough arguments for: FOO
While evaluating: (FOO 2)
1> _
```

The new listener level is identified by the new prompt, **1>**. The number used with a prompt always tells you which listener level you are on. (The Top-Level prompt is the asterisk.) If you make an error at level 1, another listener is established, with the prompt **2>**.

To return from this error level to the previous listener, use the function **clean-up-error**, which you can invoke with the keychord **Ctrl-G**. The following screen illustrates recovering from an error using **clean-up-error**:

```
* (foo 2)

ERROR:
Not enough arguments for: FOO
While evaluating: (FOO 2)
1> (clean-up-error)
Back to: Top-Level
* _
```

clean-up-error places you back at Top-Level with the asterisk prompt.

In this example, the error is straightforward enough that you probably do not need any further information to understand and correct it. However, in cases where this is not true, you can obtain information about the interrupted evalution. One way to access this stored information is with the function **backtrace** (see section 4.2, "BACKTRACE").

You can use **break** for debugging or testing your own LISP programs by including a call to **break** in your program. When **break** is called, it suspends the processing of your program and starts a new listener level, where you can perform other LISP evaluations. When you are ready, you can resume the evaluation of your program by entering the function **continue**, or the keychord Ctrl-P.

You can include a message as an argument to **break**, which prints to the screen when the break level is invoked. This message can remind you of where you are in your program, what you want to test, etc. You can also include the values of variables in the **break** message. To do this, use the ~S directive of the **format** function for each variable, and include the variable names for each ~S as separate arguments to **break**:

(break "message with a=~S and b=~S" a b)

Note that you must enclose your message in quotes and include the format arguments (e.g., **a** and **b**) in the order of their appearance in the message.

To illustrate **break**, we define the following simple program:

```
* (defun foo (a b)
      (setf a (+ a 1))
      (setf b (+ b 1))
      (break "in foo with a=~S and b=~S" a b)
      (* a b))
FOO
* _
```

This program simply adds one to the values of its two numeric arguments. The new values for the variables are then displayed in a break message. When the program continues from the break, it multiplies the new values together, returning the result.

If we apply **foo** to the numeric values 3 and 5, we obtain the following:

```
* (foo 3 5)
in foo with a=4 and b=6
1> _
```

At the new listener level you can carry out whatever evaluations you wish. If you had not included the values of **a** and **b** in the break message, you might evaluate **a** and **b**.

Once you have concluded whatever evaluations you want to perform at the break level, type **(continue)**, or **Ctrl-P**, to resume evaluation of the program. The following sample screen illustrates **continue** using the simple program from the last example:

```
* (foo 3 5)
in foo with a=4 and b=6
1> (continue)
24
* _
```

break and **continue** do not display debugging information so much as create conditions whereby debugging information can be more easily obtained. One function that obtains such information is **backtrace**.

4.2 BACKTRACE

The procedure **backtrace** displays LISP forms that have not completed evaluation. **backtrace** can be used at any time, but is most helpful at a break or error level. The keychord **Ctrl-B** may also be used to initiate a **backtrace**. If used at Top-Level, it prints only itself as the LISP form which is incomplete in its evaluation. When there is more than one[11] incomplete form, as is the case at a break or error level, the form encountered most recently prints first, the preceding form prints second, and so on. Following the display of forms, **backtrace** always returns the value **nil**.

We can illustrate **backtrace** with the same program used to illustrate **break**. We defined the program as follows:

```
(defun foo (a b)
   (setf a (+ a 1))
   (setf b (+ b 1))
   (break "in foo with a=~S and b=~S" a b)
   (* a b))
```

If we execute **backtrace** at the break level produced by **foo**, we obtain the following:

```
* (foo 3 5)
in foo with a=4 and b=6
1> (backtrace)
(BACKTRACE)
(BREAK "in foo with a=~S and b=~S" A B)
(FOO 3 5)
NIL
1> _
```

In the series of LISP forms that print to the screen, **backtrace** itself is first because it is the most recently encountered form that has not completed evaluation. The call to **break** is the next form that prints, since it is the form

----

11. There will always be at least two incomplete functions when **break** is invoked at a break level: **backtrace** and **break**.

encountered prior to **backtrace** that is incompletely evaluated.[12] **foo** is the last incompletely evaluated form that prints. It began the evaluation process that produced the break level. The last object to print is **nil** because **nil** is the value **backtrace** returns.

The procedure **backtrace** can be particularly useful in cases where there is a problem within a series of nested functions. If the most deeply nested function calls **break** or produces an error, you can then evaluate **backtrace** at the break level to see the arguments for each of the nested functions. In many situations this will help you locate the source of the problem.

----

12. The evaluation of **break** will not complete until the function **continue** is typed.

4.3 TRACE

The **trace** procedure dynamically displays the input values and the output values (i.e., the arguments and the returned values) of functions. This facility is useful when it is not clear that the interfaces between your procedures are correctly implemented.

To use **trace**, include the function you want to test as an argument. Then, each time the specified function is evaluated, its input and output values print to the screen, as in this example with the function **append**:

```
* (trace append)
T
* (append '(12)'(34))
Entering: APPEND, Argument list: ((12) (34))
Exiting: APPEND, Value: (12 34)

(12 34)
* _
```

Caution: Apply **trace** carefully to frequently-used system functions such as **first**, **rest**, and **cons**, as this can severely slow down computation time. Also, tracing the function **trace** will cause the system to loop as **trace** tries to trace itself.

You may turn **trace** off either for a particular function currently being traced, or for all functions currently being traced:

```
* (untrace append)
(APPEND)
* (trace ncons append)
T
* (untrace)
(NCONS APPEND)
* _
```

Each of these **untrace** calls returns a list of the names of the functions being turned off. **(untrace append)** turned off the trace of **append** initiated in the preceding screen. The list **(ncons append)** shows that **(untrace)** turned off the trace of **ncons** and **append**, and that no other functions were currently being traced.

4.4 STEP

The GCLISP **step** procedure allows you to view each step in the evaluation of a LISP form and control the progress of the evaluation.

To use **step**, enter it with the form in question as its argument. For example, to evaluate the form **(+ 1 (+ 2 3))** using the **step** macro, enter the following:

(step (+ 1 (+ 2 3)))

step prints the form to the screen before any evaluation takes place. With the above sample form, the screen would appear as follows:

```
* (step (+ 1 (+ 2 3)))
(+ 1 (+ 2 3))_
```

Once you have entered the **step** function, you have a series of options which allow you to continue the computation. Each time an option completes, you may again choose among them until evaluation of the entire form completes. All the options for the **step** function are executed with the cursor motion keys located at the right of the keyboard. Note: Check to be sure that the **NumLock** key has not been pressed to shift the cursor motion keys to numeric keypad. If it has, press it again to undo the effect.)

If you are not sure what option you want or cannot remember what all of the options are, you can type '?' and a list of the options will appear on the screen as follows:

```
* (step (+ 1 (+ 2 3)))
(+ 1 (+ 2 3)) <?>
STEP commands are:
      arrow-dn   ==> Step to next level down
      arrow-rt   ==> Value of this form
      arrow-up   ==> Step to next level up
      arrow-lt   ==> PrettyPrint this form
      Ctrl-Break ==> Enter Break Level
         END     ==> Complete without more Stepping

 (+ 1 (+ 2 3))_
```

4.4.1 The arrow-dn Option

The option invoked with arrow-dn (the down-arrow key) proceeds through evaluation with the smallest sub-forms. If the current form (i.e., the one last printed to the screen) is such a sub-form, it is evaluated, and the next form prints to the screen. If the current form can be divided into further sub-forms, the next smallest sub-form prints to the screen.

If we use just the arrow-dn option for evaluating the sample form **(+ 1 (+ 2 3))**, the response is as follows:

1. The first sub-form **1** prints to the screen with the first execution of arrow-dn.

2. With the second execution of arrow-dn, the first sub-form **1** evaluates (since this form cannot be divided into any further sub-forms) and the next form, which is **(+ 2 3)**, prints to the screen.

3. Since **(+ 2 3)** divides into sub-forms, the first sub-form within **(+ 2 3)**, which is **2**, prints on the third execution of arrow-dn.

4. On the fourth execution, the form **2** evaluates and the form **3** prints, as the next sub-form within **(+ 2 3)**.

5. On the next evaluation of arrow-dn, the form **3** evaluates. Since this is the last sub-form of **(+ 2 3)**, **(+ 2 3)** also evaluates; since **(+ 2 3)** is the last form in the overall form, the overall form evaluates too.

The screen display for this process is as follows:

```
* (step (+ 1 (+ 2 3)))
(+ 1 (+ 2 3)) <arrow-dn>
 1 <arrow-dn>
  1 = 1
 (+ 2 3) <arrow-dn>
  2 <arrow-dn>
   2 = 2
  3 <arrow-dn>
   3 = 3
  (+ 2 3) = 5
 (+ 1 (+ 2 3)) = 6
6
* _
```

4.4.2 The arrow-rt Option

The arrow-rt option evaluates the current form (i.e., the one last printed to the screen) and prints the next form onto the screen.

At the beginning, the entire form is the current form. If we choose arrow-rt as the first option, the entire form is evaluated and the return value prints to the screen. If we begin instead with the arrow-dn option, which prints the form 1 to the screen, and then choose the arrow-rt option, it evaluates the form 1 as the current form and prints the next form to the screen. If we again select the arrow-rt option, it evaluates the current form (i.e., (+ 2 3)), and because it is the last form in the overall form, the evaluation for the entire form prints to the screen too.

Here is how the screen looks in response to the sequence of options just discussed:

```
* (step (+ 1 (+ 2 3)))
(+ 1 (+ 2 3)) <arrow-dn>
 1 --> 1 <arrow-rt>
 (+ 2 3) --> 5 <arrow-rt>
 (+ 1 (+ 2 3)) = 6
6
* _
```

The initial arrow-dn option prints the form 1 that appears directly below the printing of the entire form. The evaluation of 1 (represented by the arrow to its right and the 1 to the right of the arrow) and the printing of the next form (i.e., (+ 2 3)) occurs with the first execution of the arrow-rt option. When this option is chosen again, it evaluates the current form (i.e., (+ 2 3)) and, because it is the last form in the overall form, evaluates the entire form too.

4.4.3 The arrow-up Option

Arrow-up evaluates the current form (i.e., the one printed on the screen) and the enclosing form.

If we again start with the arrow-dn option and then continue with the arrow-up option, first arrow-dn prints the form **1**, then arrow-up evaluates **1** (the current form) and **(+ 2 3)** (the next form). This completes evaluation of the entire form, which prints to the screen. The following sample screen illustrates:

```
* (step (+ 1 (+ 2 3)))
(+ 1 (+ 2 3)) <arrow-dn>
 1 <arrow-up>
(+ 1 (+ 2 3)) = 6
6
* _
```

4.4.4 Other Options

There are three other options with the **step** function not specifically associated with evaluation. One of these, arrow-lt, pretty prints the current form (i.e., prints it again in a human readable format; see section 4.5, "PPRINT," for an explanation of pretty printing).

Another option, **Ctrl-Break**, establishes a new listener level. At the listener, the following variables are available for evaluation: **step-form**, which is bound to the current form; **step-values**, which is bound to the values list returned from the stepped evaluations completed thus far; and **step-value** (without the "s"), which is bound to **(first step-values)**.

The following sample screen shows the **Ctrl-Break** option used after two executions of the arrow-dn option. At the listener, each of the special variables for this option is evaluated.

```
* (step (+ 1 (+ 2 3)))
(+ 1 (+ 2 3)) <arrow-dn>
 1 <arrow-dn>
  1 = 1
 (+ 2 3) <Ctrl-Break>
STEPPER BREAK
1> step-form
(+ 2 3)
1> step-values
(1)
1> step-value
1
1> (continue)
Back to STEP with form:
(+ 2 3)_
```

Notice that **step-values** in this case returns a list of only one value (the value of **step-value**). This is because the previous form (i.e., 1) did not return multiple values.

The last option, **end**, turns off evaluation by steps and causes the entire form to be evaluated. The following sample screen shows the **end** option after an initial execution of the arrow-dn option.

```
* (step (+ 1 (+ 2 3)))
(+ 1 (+ 2 3)) <arrow-dn>
 1 <end>

6
* _
```

4.5 PPRINT

The pretty printer displays text in an easily-read format. It enables you to analyze components of a LISP function more easily. Suppose you have entered this function definition:

```
* (defun foo (a &optional b c)
  (do ((x a (+ 1 (first b)))
       (y b (rest b))
       (z c (rest c)))
      ((null y) (print "stopped"))
    (print 1)
    (print 2)
    (print 3)))
FOO
* _
```

The function **symbol-function** displays the function definition of **foo** with no regard for the program structure:

```
* (symbol-function 'foo)
(LAMBDA (A &OPTIONAL B C) (DO ((X A (+ 1 (FIRST B)))
(Y B (REST B))(Z C (REST C)))((NULL Y)(PRINT "stopped"))
(PRINT 1)(PRINT 2)(PRINT 3)))
* _
```

For a clearer representation, use the **pprint** function:

```
* (pprint (symbol-function 'foo))

(LAMBDA (A &OPTIONAL B C)
  (DO ((X A (+ 1 (FIRST B)))
       (Y B (REST B))
       (Z C (REST C)))
      ((NULL Y)
       (PRINT "STOPPED"))
    (PRINT 1)
    (PRINT 2)
    (PRINT 3)))
* _
```

4.5.1 Formatting Rules Used with PPRINT

The GCLISP **pprint** function prints objects in accord with the following set of rules.

1. Individual numbers and symbols print just as they do with the ordinary **prin1** function.

2. Lists have various formats depending on the first element of the list. If the first element is a symbol, then **pprint** looks at its **pprint** property, which determines how the list will pretty print.

3. When there is no value associated with **pprint** on the symbol's property list (i.e., when **(get (first list) pprint) => nil**), then **pprint** assumes that the list has no special format requirements and prints it on a single line if possible. If the list will not fit on one line, then each element prints on a separate line, all indented the same number of spaces.

4. If the value of the **pprint** property is a symbol, the function **pprint** assumes the symbol names a function, which it calls to print the list. When **pprint** calls this function, it passes its argument list to it. The following sample screens illustrate the process of:

   - assigning the name of a print function to the pprint property of a symbol;

   - defining that print function; and

   - pretty printing a list whose first element has as the value of its **pprint** property the defined print function.

   First the **pprint** property for a symbol **foo** is set to the value **foo-pprinter**:

   ```
   * (setf (get 'foo :pprint) 'foo-pprinter)
   FOO-PPRINTER
   * _
   ```

   This causes the function **pprint** to call **foo-pprinter** any time its argument is a list whose first element is the symbol **foo**. **foo-pprinter** then prints the list that begins with **foo**.

The function **foo-pprinter** is defined as follows:

```
* (defun foo-pprinter (object)
    (prin1 'foo)
    (dolist (I (rest object))
      (terpri)
      (prin1 I))
  )
FOO-PPRINTER
* _
```

First **foo-pprinter** calls **prin1** to print "foo." Then **dolist** is called and isolates successive elements of the list represented by *object*, which consists of the arguments **pprint** passes to **foo-pprinter.** For each element, the function **terpri** ("terminate print") sends a Newline character, so that the element is printed on a new line by **prin1.**

Thus, if we pretty print the list **(foo 1 2 3)**, the result is as follows:

```
* (pprint '(foo 1 2 3))
FOO
1
2
3
NIL
* _
```

pprint calls **foo-pprinter**, which prints **foo** and then prints each of the other elements in the list on successive lines. Finally, the **pprint** function returns the value **nil.**

Thus, using a function name as the value of the pprint property of the first element of a list enables you to control how **pprint** formats the printing of the list. You can define formatting routines for special lists, or even completely redefine the **pprint** facility.

5. If the value of the **pprint** property is not a symbol, it must be a list (called a *template*) that provides control information for the system-supplied **pprint** function. The template is really a list of sub-lists, with each sub-list controlling a separate component of the form in question (i.e., the argument to **pprint**). For example, the **do** special form is composed of three parts: The *iterators*, the *termination clause*, and the *body* of the **do**. The symbol **do** contains on its property list an entry for the **pprint** property as follows:

   ((do-bindings 5) (prog-body 5) (prog-body-rest 2 T))

   The keywords

   do-bindings, *prog-body*, and *prog-body-rest*

   specify the display for the first, second, and remaining sub-forms of **do**. The numbers associated with each keyword specify the number of characters indented for each sub-form.

Note that the file \LISPLIB\PPRINT.LSP, provided in your GCLISP package, contains a detailed specification of the variables and functions available to user-defined **pprint** functions. This file includes the full specification of the keywords for templates, as well as a list of all forms which pprint supports. Please refer to this file for information needed to modify and extend the GCLISP pretty print facility.

Chapter 5

An Application: The PIANO Program

Now that you have some familiarity with the GCLISP environment, you are ready to build GCLISP applications. In this chapter we present the development of a sample GCLISP program, which you can use as a model to get started.

For this sample application, we choose a program that alters the function of several keyboard characters, because this type of program has a general usefulness. Even though you may not have particular interest in the application developed here, it is likely that you will eventually want to alter the functions assigned to keyboard characters.

The program we present here turns the PC keyboard into a piano keyboard. To sidestep the difficult hardware interface required for this program, we start with certain GCLISP functions that produce elements of music. Discussion of these functions (and the hardware interface they require) also appears in this chapter, but after the general discussion of the program has concluded. This way, you may choose not to read it without having to skip pages. Finally, we have tried to orient the discussion toward ideas that may help you in developing GCLISP applications.

Note: After reading this chapter, you can invoke the PIANO program by calling the function **piano**.

5.1 Elements of the Piano Keyboard Program

To build a program that defines the computer keyboard as a piano keyboard, we must call a routine that plays notes each time certain keyboard characters are typed. From this functional description we can identify three elements that we need for our piano keyboard program:

1. a routine that plays musical notes;

2. a mapping of keyboard characters to musical notes; and

3. a program structure that reads keyboard characters and calls the music routine to play the note mapped to that particular keyboard character.

The first of these three components to our program is provided through a function called **play**, which takes three different kinds of arguments:

- keyword designations for notes (e.g., :C for the musical note C);

- octave values that raise and lower the octave in which the notes play; and

- time values for the duration a note plays.

We analyze the structure of **play** at the end of this chapter (sections 5.2.6 - 5.2.7). For now we concentrate on defining the second and third elements of the piano keyboard program.

5.1.1 Mapping Keyboard Characters to Notes

To define the terminal keyboard in a way that approximates a piano keyboard, we can pick two rows of keys: one for the whole tones (the white keys on the piano) and one for the half tones (the black keys on the piano). Further, we can let the upper row of keys -- the ones closer to the top of the keyboard -- represent the black keys, and the lower row of keys represent the white keys. This way, the terminal keys representing the black piano keys are both in-between and recessed from the terminal keys representing the white keys, as on a piano.

Since there are not 88 keys on the computer keyboard, as there are on a piano keyboard, we need to define a particular set of keys on the computer keyboard that can be used to play all (or most) of the notes on a piano. For this we have recourse to the twelve notes of the conventional musical scale. A scale provides an appropriate subset of notes, because the piano keyboard is such that any row of twelve keys plays one full scale. Each scale of twelve notes is exactly one octave higher or lower than the one next to it. Therefore we can use twelve notes and a set of octave values to cover the range of notes on a piano. That is, we can raise or lower the octave value of any of the twelve notes on a scale so as to play any of the eighty-eight keys on the piano. For instance, if we define the note C of our scale as middle C on the piano, we can change the octave value to play the other C notes on the piano keyboard.

If we start by mapping note C of the scale to the A-key on the keyboard, mapping C# to the W-key, and so on moving up the scale and across the keyboard from left to right, our piano keyboard will have the correlation between notes and keyboard characters shown in Figure 1.

KEY		NOTE
A	--->	C
W	--->	C#
S	--->	D
E	--->	D#
D	--->	E
F	--->	F
T	--->	F#
G	--->	G
Y	--->	G#
H	--->	A
U	--->	A#
J	--->	B
K	--->	C

FIGURE 1. The Mapping Between Keyboard Keys and Musical Notes

5.1.2 Reading Keyboard Characters

Now that we have defined the keyboard as a piano keyboard, we can proceed to the third element of the program: developing a program structure that reads keyboard characters and calls the **play** function to produce the note associated with that character. Since this really involves two steps -- reading the characters and calling the **play** routine -- we can treat each separately.

We begin with reading characters from the terminal. To read a character from the keyboard, we can use the **read-char** function. Since we will want to read every keystroke to the program (assuming only keys that have been defined as notes will be pressed), we set up a loop that repeatedly reads a character.

The GCLISP function for building such a loop is **do**. The syntax for **do** can sometimes be complex, but for our purposes it can be relatively simple. Since the initial function call is **read-char** and the successive function calls are also **read-char**, all we need do to set up a **do** loop is pick a variable name for the character read by **read-char**. If we use the symbol **char** as this variable name, then our **do** routine looks like this:

```
(do ((char (char-upcase (read-char))
           (char-upcase (read-char))))
    (...)...)
```

This form reads a character from the terminal and assigns its upper-case value to the variable **char**.

The meta-form **(...)** is for the *end test* for the **do** loop. For the time being, we leave this test out. The three dots and the final parenthesis indicate that the body of the **do** is also unspecified as yet.

5.1.3 Representing Keyboard Characters in ASCII Code

When the GCLISP function **read-char** returns the character it reads, it transforms it into ASCII code form. This means that the value assigned to the variable **char** (each time a keyboard character is read) is the ASCII representation of that character. Because the value of **char** needs to be matched with another character, that other character has to be in ASCII format also.

Instead of looking up ASCII code for characters to do this, we can use the sharp-sign-backslash macro (#\). When this macro precedes an alphanumeric character, it signals the GCLISP reader to produce the ASCII code for the character. That is, to represent the character A in ASCII code, we write:

```
#\A
```

5.1.4 The Program Structure for Calling the PLAY Routine

To formulate the basic program structure for our piano keyboard, we need to be able to call the function **play** with the appropriate note or octave change for each keystroke. For this, we need to set up a conditional structure that tests which keyboard character was struck and invokes the **play** routine with the appropriate argument (note, octave, etc.).

Remembering from our **do** loop that the symbol **char** represents keyboard characters, we write a conditional statement that calls the **play** routine when A is struck on the keyboard:

```
(cond ((eq char #\A)  (play :c))...)
```

This condition specifies that the note C plays whenever the **read-char** function returns the ASCII code for the character A.

To shift the octave value, we can call the **play** routine with the argument **:od** to lower the octave and the argument **:ou** to raise the octave. If we choose the character "-" to lower the octave value, the conditional expression looks like this:

```
(cond ((eq char #\-) (play :od))...)
```

This condition shifts the octave down each time the minus key (-) is pressed. No note plays -- only the octave for the next note shifts down. We can write an analogous expression for raising the octave using the character "+".

If we put together the conditional structures we have just developed with the **do** loop into a single program structure defining the function **piano**, we get something like this:

```
(defun piano ()
  (do ((char (char-upcase (read-char))
             (char-upcase (read-char))))
      (...)
    (cond  ((eq char #\A)  (play :c))
           ((eq char #\W)  (play :cs))
           ((eq char #\S)  (play :d))
           ((eq char #\E)  (play :ds))
           ((eq char #\D)  (play :e))
           ((eq char #\F)  (play :f))
           ((eq char #\T)  (play :fs))
           ((eq char #\G)  (play :g))
           ((eq char #\Y)  (play :gs))
           ((eq char #\H)  (play :a))
           ((eq char #\U)  (play :as))
           ((eq char #\J)  (play :b))
           ((eq char #\K)
            (play :ou)
            (play :c)
            (play :od))
           ((eq char #\-)  (play :od))
           ((eq char #\+)  (play :ou)))
    (speaker :off)))
```

<u>Note</u>: S represents a sharp sign in the notation for the musical notes. Thus, **:as** stands for A#.

5.1.5 Putting in an End Test

Aside from the note keys, we also have to establish an exit key. (Remember that we left the end test for the do form incomplete.) If we choose the character "X" for exit, then the end test for the program would be as follows:

```
(eq char #\X)
```

and the entire piano program looks like this:

```
(defun piano ()
  (do ((char (char-upcase (read-char))
             (char-upcase (read-char))))
      ((eq char #\X)
       "Nice tune!")
    (cond  ((eq char #\A)  (play :c))
           ((eq char #\W)  (play :cs))
           ((eq char #\S)  (play :d))
           ((eq char #\E)  (play :ds))
           ((eq char #\D)  (play :e))
           ((eq char #\F)  (play :f))
           ((eq char #\T)  (play :fs))
           ((eq char #\G)  (play :g))
           ((eq char #\Y)  (play :gs))
           ((eq char #\H)  (play :a))
           ((eq char #\U)  (play :as))
           ((eq char #\J)  (play :b))
           ((eq char #\K)
            (play :ou)
            (play :c)
            (play :od))
           ((eq char #\-)  (play :od))
           ((eq char #\+)  (play :ou)))
    (speaker :off)))
```

5.1.6 Modifying and Revising the PIANO Program

There are several types of things we could do to improve our program. We could, for instance, add to the ease with which other people could use it. For example, we might write someplace on the screen that "X" is the exit key.

We might otherwise wish to modify our function to give ourselves greater flexibility. For example, instead of hard-coding the duration value for the notes, we could include conditional statements for tempo values, just as we do now for notes and octaves.

Another type of modification is in programming style. We could improve upon the elegance of our program by using the **case** special form instead of the **cond** special form. The next section, which explains various music functions, can help you make some of these modifications.

5.2 Musical Functions and Variables

The GCLISP program **piano** transforms the computer keyboard into a piano keyboard. **piano** calls the function **play** in order to carry out the actual playing of notes. The source code for **piano**, **play**, and their subordinate functions may be found in the file \EXAMPLE\MUSICPGM.LSP. This section gives explanations of each of the functions and variables used to implement the **play** function. (Several of these functions involve the hardware interface necessary for providing the elements of music. For more information regarding hardware features of the IBM PC, consult the IBM PC Technical Reference Manual.)

5.2.1 Musical Global Variables

The following function calls establish a series of global variables and constants for the GCLISP music environment:

```
(defvar *music-octave* 2)

(defvar *music-scale*
        '(:c 494 :cs 466 :d 440 :ds 415 :e 392 :f 370
          :fs 349 :g 330 :gs 311 :a 294 :as 277 :b 262))

(defvar *music-time* 5)

(defconstant speaker-control-port #x61)

(defconstant timer-select-port #x43)

(defconstant frequency-set-port #x42)
```

The three defined variables -- ***music-octave***, ***music-scale***, and ***music-time*** -- represent values that define aspects of music. ***music-octave*** and ***music-scale*** together define the pitch or frequency value for a note, and ***music-time*** is used to define tempo, or time value for a note.

You can think of these three variables as the three components of a note. The three constants represent the mechanics of actually producing sound:

- ***music-scale*** defines twelve notes by associating keywords (:C, :D, :E, etc.) with integer values that produce the frequencies for the notes represented by the keywords. (Note: The integer values themselves are not the frequency values for the notes. Rather, they modify a standard frequency generated by the timer chip to produce the scale frequencies. See the **sethertz** function description in section 5.2.3 below.)

- ***music-octave*** represents an octave value. It is used as a parameter for the **lsh** function to change the frequency value of a note to one octave higher or lower.

- ***music-time*** refers to the duration a note sounds. The ***music-time*** value you give to a quarter note (for 4/4 and 3/4 time) establishes a tempo. The unit of duration is defined by the **sleep** function discussed below. The time it takes the GCLISP interpreter to evaluate a single empty **dotimes** loop is the value for ***music-time*** represented by the integer 1.

You can think of **speaker-control-port, timer-select-port,** and **frequency-set-port** as components of an instrument that plays music. These three components all define IBM-PC specific, 8-bit ioports which provide program interfaces to hardware features of the PC.

5.2.2 The OCTAVEMOVE Function

```
(defun octavemove (action)
  (case action
    (:ou
     (decf *music-octave*))
    (:od
     (incf *music-octave*))
     ))
```

This function raises or lowers the current value of the global variable ***music-octave*** by 1. If a value **:ou** is given for the parameter **action**, the value for the variable decreases by one; if the value of **action** is **:od**, the variable value increases by one. Thus, this function enables new notes to play an octave higher or lower than the current octave. (Note: When the value of ***music-octave*** decreases, the next note plays an octave *higher* (and vice versa).)

5.2.3 The SETHERTZ and SPEAKER Functions

These functions control the mechanics of actually producing musical notes. **speaker** turns on or off the speaker, which allows individual notes to sound. **sethertz** controls the frequency generator used to produce notes. Both functions utilize the **%ioport** primitive (discussed below).

The **sethertz** function is as follows:

```
(defun sethertz (hertz)
  (%ioport timer-select-port #x0B6 nil)
  (%ioport frequency-set-port (logand hertz #x0FF) nil)
  (%ioport frequency-set-port (lsh hertz -8) nil)
  )
```

This function sends an integer value to ioport **frequency-set-port** in order to generate the frequency for a note. The note frequency is equal to the frequency of the timer chip divided by the integer sent to the ioport. The **hertz** parameter represents integer values that divide into the value of the timer chip frequency to produce the frequencies for musical notes. The integer values defined by the global variable ***music-scale*** provide a set of such **hertz** values for the notes of a scale.

To understand the **sethertz** function in greater detail, we need to understand the **%ioport** primitive. This primitive has three parameters:

- the ioport address (e.g., **frequency-set-port**, **timer-select-port**)

- the ioport data value (e.g., **#x0B6**)

- the indicator for a 16 bit ioport data value (e.g., **nil**)

Essentially, the **%ioport** primitive sends the ioport data value to the ioport address. The primitive can only send 8 bits at a time, so the third parameter (the indicator of a 16 bit data value) should always be **nil**.

The **sethertz** function, then, sends an **ioport** data value (**#x0B6**) to **timer-select-port** that opens that ioport. After the ioport **timer-select-port** is open, the integer value for a new note is sent to ioport **frequency-set-port** as the ioport data value.

Because integer values are 16 bits and the **%ioport** function only sends the low-order 8 bits, sending the integer requires two executions of the **%ioport** function. First, the low-order 8 bits of the integer are sent, by masking the upper 8 bits using the **logand** function and the mask **#x0FF**. Second, the upper 8 bits of the integer are sent by right-shifting them into the region of the lower 8 bits.

The note frequency produced by **sethertz** can only sound, however, if the speaker is on. The function that turns the speaker on and off is as follows:

```
(defun speaker (switch
                &aux (val (%ioport speaker-control-port
                                   nil
                                   nil)))
  (case switch
    (:on
     (%ioport speaker-control-port (logior val 3)  nil))
    (:off
     (%ioport speaker-control-port (logand val #x0FC) nil))
     ))
```

The parameter **switch** accepts the values **:on** or **:off**. **:on** sets the low-order two bits of **speaker-control-port** on (without affecting the other six bits). **:off** clears these two bits (without affecting the other six bits).

The **sethertz** and **speaker** functions are used in the definition of the **beep** function in section 5.2.5 below.

5.2.4 The SLEEP Function

```
(defvar *tempo* 1)

(defun sleep (time)
  (dotimes (i time)
  (dotimes (j *tempo*) (dotimes (k 1000)))
  ))
```

This function sets up a wait loop that defines the duration of a note. Actually, **sleep** defines three nested loops. The innermost loop is an empty loop that iterates one thousand times. The intermediate loop repeats the number of times set by the variable ***tempo***. And the outer loop iterates the number of times represented by **time**.

You might test a value of 1 for **time** as the length of a sixteenth note, 2 the value of an eighth note, and so on. For slower pieces and faster pieces the values for a given type of note (quarter, eighth, etc.) would increase and decrease, respectively.

5.2.5 The BEEP Function

```
(defun beep (tone time)
  (sethertz tone)
  (speaker :on)
  (sleep time))
```

This function plays a note by putting together the function that produces a frequency for a note, the function that defines a duration period, and the function that turns on and off the speaker. **sethertz** produces the frequency for the note, which sounds for the time duration produced by the evaluation of **sleep**. **beep** does not turn the speaker off, thus allowing the caller to either change the tone or turn off the speaker.

5.2.6 The PLAY Function

```
(defun play (music &optional (time *music-time*))
  (if (numberp music)
      (setq *music-time* music)
      (case music
            (:r    (sleep time))
            (:ou   (octavemove  :ou  ))  ;octave UP
            (:od   (octavemove  :od  ))  ;octave DOWN
            (otherwise
             (let ((freq (getf *music-scale* music)))
               (when (null freq)
                   (error "Unknown frequency: ~s" freq))
               (beep (lsh freq *music-octave*)
                *music-time*))
                   ))
        ))
```

play does one of four different things:

1. It raises or lowers the octave within which a note plays;
2. It resets the duration for the note;
3. It plays (another) note; or
4. It rests.

The parameter **music** governs how **play** behaves:

1. If the value of **music** is **:ou** or **:od**, then **play** changes the octave;

2. If the value is a number, then **play** resets the duration;

3. If the value ie a note (e.g., **:c** or **:d**), then **play** plays that note;

4. If the value is **:r**, then **play** rests for the duration.

The optional parameter **time** temporarily overrides the duration used for playing a note. If **time** is omitted, the specified note plays for duration ***music-time***.

This function puts together the functions we have discussed already: **octavemove**, **sleep**, and **beep**.

5.2.7 The PLAYLIST Function

Like **play**, **playlist** puts together functions already built. **playlist** utilizes **play** in a **dolist** loop to play a sequence of notes:

```
(defun playlist (notelist)
  (dolist (note notelist) (play note))
  (speaker :off))
```

The **notelist** parameter includes the same three elements used as arguments to **play**: notes (**:c**, **:d**, **:e**, etc.); octave changes (**:ou** and **:od**); and time values. The **dolist** loop evaluates **play** for each element of **notelist**.

5.2.8 Putting Together Music Programs

One way to think of composing music is as the putting together of notes into phrases which are repeated in variation. You can implement this technique for musical composition by using **playlist** to create phrases and lines of notes and then putting these lines together. For instance you could have one function composed of several executions of **playlist** using the following format:

```
(defun music ()
  (playlist (...))
  (playlist (...))
  (playlist (...))
   ...)
```

You could also use **dotimes** loops to repeat phrases defined by **playlist**. For example:

```
(dotimes  (i 3) (playlist  '(5 :gs :e :gs :e)))
```

You can put together these **dotimes** loops into functions and put those functions together as programs (or larger composite functions) and in this way build musical compositions in the same step-by-step, component-by-component fashion used to develop the music functions themselves.

Index

GOLDEN COMMON LISP
REFERENCE MANUAL

Version 1.01

Table of Contents

Chapter 1

Introduction

GOLDEN COMMON LISP (or more briefly, GCLISP) is a dialect of COMMON LISP designed to work on a variety of processors, including those found in commercial microcomputers such as the **IBM PC (TM)**.

1.1 Purpose

GCLISP was designed with the following goals in mind (not in order of importance):

Commonality	GCLISP is designed according to the COMMON LISP core specification. COMMON LISP is intended to serve as a common dialect, shared by many different implementations.
Portability	GCLISP programs which restrict themselves to those features specified as part of the COMMON LISP core may be easily transported to other COMMON LISP implementations. In addition, the GCLISP environment is designed to be easily transported to various host environments.
Power	GCLISP attempts to provide the most powerful features of COMMON LISP while leaving out those features which are of limited usefulness. At the same time, powerful concepts found in other LISP dialects (e.g., ZETALISP's stack groups), but which are not (yet) part of COMMON LISP, have been included. GCLISP also provides a complete interface (both low and high level) to the host hardware and operating system.
Expressiveness	Although GCLISP does not provide every feature specified in COMMON LISP, most of the omitted

features can be easily defined in GCLISP.

Compatibility	GCLISP is a compatible subset of the COMMON LISP core specification. It also incorporates various ZETALISP concepts.
Efficiency	In order to reduce the processing power and memory demands on the programmer, LISP puts great demands on the processing power and memory of the computer. Therefore, efficiency was one of the primary concerns in the design and implementation of GCLISP.
Stability	GCLISP will evolve toward full implementation of the COMMON LISP standard. Software designed with the COMMON LISP specification in mind will be compatible with future versions of GCLISP.

This document is a language reference manual. As such, its basic purpose is to specify the syntax and semantics of the various language constructs. It is not intended to be a language tutorial nor a system users' guide. Therefore, it addresses itself to the intended practical use of a particular construct only to the degree that such a description may elucidate its semantics.

Readers of this manual should have a good understanding of programming in general and LISP in particular. Those who want to learn how to program in LISP should turn to the book LISP (Winston and Horn, 1984) which is included in the GCLISP package. Those who want information on the actual use of GCLISP should turn to the GCLISP Users' Guide.

This manual is designed to be used in conjunction with the COMMON LISP Reference Manual (Steele, 1984) (hereafter referred to as the **CLRM**). Therefore, this manual adopts, as much as possible, the format and notational conventions of the **CLRM**. In fact, this manual uses the same chapter, section, and subsection numbering as the **CLRM**.

Many of the features described in this manual are described at greater length in the **CLRM**. Readers who are totally unfamiliar with COMMON LISP may find it helpful to peruse the **CLRM** before reading this manual.

Many of the entries in this manual are also available via GCLISP On-Line Help. Because of this, some of the entries may repeat information provided in other entries.

This manual was written with the following goals in mind (in order):

Precision	Precision is necessary for two reasons. First, this manual is responsible for

specifying the *exact* behavior of every GCLISP entity. Secondly, when GCLISP diverges from the COMMON LISP specification, it often does so in ways which would not be apparent in an informal description.

Clarity — Hopefully, this is a self-explanatory goal. It is secondary to precision since this is a language reference manual, not a language tutorial. In a language tutorial, precision is secondary to clarity.

Concision — Because most of the features of GCLISP are described in depth in the **CLRM**, this manual is designed to be a concise summary of the **CLRM**. In addition, since much of this document is accessible via GCLISP On-Line Help, brevity is of practical concern.

Readers are strongly encouraged to suggest areas in which this manual falls short of these goals. A comment card is included in the GCLISP package for this purpose.

1.2 Notational Conventions

The notational conventions used in this manual are, as much as possible, identical to those used in the **CLRM**. The next section provides a brief summary of the **CLRM**'s conventions and this manual's variations.

1.3 This Manual's Conventions

1.3.1 Description of Values

In the **CLRM**, the first line of function, macro, special form, and variable entries specifies the name and any arguments of the entity. This manual adds a description of the values returned by the entity. Figure 1 illustrates a typical entry.

Figure 1: Sample Function Entry

[*Function*]

sample-function *integer1 integer2* => *sum difference*

This function returns the *sum* and the *difference* of *integer1* and *integer2*.

As the example shows, the result of the function call is indicated by an evaluation arrow (=>) followed by one or more names which describe the returned values. (The first line of the description of a function or special form which does not return any values (e.g., **go**) does not contain an evaluation arrow.)

The names of results are intended to be as descriptive as possible. The following list describes the result naming conventions:

boolean — The result name *boolean* refers to a result which may be either true (**t**) or false (**nil**).

result — Result names which contain the word *result* indicate that only a single value is returned.

results — Result names which contain the word *results* indicate that multiple values may be returned.

last-form — Result names which contain the word *last-form* indicate that the results of the last (i.e., rightmost) subform are returned. Forms containing an *implicit* **progn** typically have this type of result.

last-evaled-form — Result names which contain the word *last-evaled-form* indicate that the results of the last (i.e., rightmost) subform which was evaluated are returned. Control structures such as **case** and **cond** typically have this type of result.

nil/... — Result names with the prefix *nil/* indicate that either **nil** or some other result will be returned. In general, a / separates alternative results.

1.3.2 Capitalization in Special Form and Macro Call Descriptions

Special forms and macro calls are more difficult to describe than function calls since their syntactic components may or may not be evaluated. To lessen this confusion, this manual adopts the following typographic convention:

In the first line of macro and special form entries, the syntactic components which are never evaluated are capitalized, while components which may or may not be evaluated (e.g., the subforms in the special form **and**), are in all lower case (just like function parameter names).

For example, the first line of the entry for the **setq** special form looks something like Figure 2:

Figure 2: First Line of setq Entry

[*Special form*]

setq {*Symbol form*}* **=>** *last-form-result*

The component *Symbol* begins with a capital letter since it is never evaluated. On the other hand, the component *form* is always evaluated, so it is in all lower case.

1.3.3 Notes

As in the **CLRM**, this manual defines two special types of notes: **Compatibility notes** and **Implementation notes**.

In the **CLRM**, a **Compatibility note** points out where COMMON LISP is either particularly compatible or incompatible with its predecessors; while in this manual, a **Compatibility note** always points out where GCLISP differs from the COMMON LISP core specification.

An **Implementation note** in the **CLRM** suggests possible implementation strategies; while in this manual, an **Implementation note** points out the particular implementation strategy used in GCLISP.

1.3.4 List of Conventions

The following list summarizes the typographical and notational conventions used in both this manual and the **CLRM**. For more detailed explanations of the various conventions, see Chap. 1 of the **CLRM**.

entity-name The names of all functions, special forms, macros, global variables, and named constants appear in the same typographical style as **entity-name**.

parameter-name
: The names of all function parameters and the names of special form and macro components appear in the same typographical style as *parameter-name*.

(example-function 5 'foo)
: All examples of actual code appear in the typographical style of **example-function**.

=>
: This sign appears between a *form* and its *values*, indicating that the evaluation of *form* results in *values*.

==>
: This sign appears between a macro-call form and its *expansion*.

<=>
: This sign appears between two forms, indicating that they are semantically equivalent. In other words, the evaluation of one of the forms results in the same values and side effects as the evaluation of the other form.

[...]
: Brackets enclose an optional component in the description of special forms and macros.

{...}*
: Braces with a trailing asterisk enclose a component which may appear zero or more times. This convention is used in the description of special forms and macros.

{...}+
: Braces with a trailing plus-sign enclose a component which may appear one or more times. This convention is used in the description of special forms and macros.

|
: Within braces, the vertical bar separates mutually exclusive alternatives.

(*first* . *rest*)
: The *dotted-list* notation is used in some examples. The dot informs the reader that *rest* denotes the remaining elements (i.e., the **rest** or **cdr**) of the list, not the last element.

(...)
: Parentheses delimit a list of elements. Lists may contain any number of elements of any type (including lists).

'
: The single quote (also known as an accent acute or an apostrophe) precedes an object which is not intended to be evaluated. Thus, '*object* is an abbreviation for (quote

object).

;	The semicolon precedes a comment (which extends to the end of the line). Comments are ignored by the LISP reader; their sole purpose is the enlightenment of the human reader.
"..."	Double quotes delimit character strings.
\	The backslash character is a *single escape* character. The character which it precedes loses any special significance it may have to the LISP reader; it is treated as an ordinary letter.
|...|	Vertical bars delimit symbols whose print-names are to be taken literally. The vertical bar is a *multiple escape* character. Such names may contain special characters (e.g., parentheses, whitespace). Note that a single vertical bar used in a macro or special form description has a different meaning.
#'	The number sign (also known as the sharp sign, the pound sign, the hash mark, and the oglethorpe) followed by a single quote precedes an object which names a function. The evaluator does not evaluate the object, rather it returns the function named by the object. Thus **#'***object* is an abbreviation for **(function** *object***)**.
#\	The number sign followed by a backslash precedes a character or a character name (e.g., **Tab**) which is to be read as a character object.
#(...)	Parentheses preceded by a number sign enclose the elements of a simple general vector. (The elements of the vector may be of any type.)
#x	The number sign followed by the letter **x** precedes a number in hexadecimal (i.e., radix-16) notation.
#o	The number sign followed by the letter **o** precedes a number in octal (i.e., radix-8) notation.
#b	The number sign followed by the letter **b** precedes a number in binary (i.e., radix-2) notation.
:	The colon character is a package marker. The name preceding it is the name of a package,

while the name following it is the name of a symbol in that package. If no name precedes the colon then the name following the colon is a *keyword*.

1.3.5 Conventions Used in Examples

The examples of code which appear throughout the manual are primarily intended to demonstrate the counter-intuitive effects or results of a given function, macro, or special form.

All examples consist of a single form (which may contain more than one subform) followed by either the evaluation arrow (=>) and the resulting values or some text describing what action is taken (e.g., *signals an error*).

All symbols (other than those which name predefined functions, variables, etc.) used in the examples (e.g., foo, bar) are intended to be unbound, to have no function definition, and to have an empty property list.

Every effort was made to keep the number of auxiliary functions, special forms and macros to a minimum, so that the point of an example would not be obscured by an unfamiliar supporting function.

The following is a list of the special forms, macros, and functions (other than the entity being explained of course) which are used extensively throughout the examples:

*****	**+**	**-**	**<**	**=**
>	**and**	**append**	**car**	**cdr**
cons	**defun**	**first**	**float**	**gensym**
if	**incf**	**lambda**	**let**	**list**
member	**not**	**null**	**progn**	**setf**
setq	**unless**	**values**	**when)**	

If the reader is familiar with most of these, the examples should be easily understood.

Chapter 2

Data Types

2.1 Numbers

[*Type*]

number

COMMON LISP defines three subtypes of **number**: **rational**, **float**, and **complex**.

Compatibility note: GCLISP currently supports two subtypes of **number**: **fixnum** (a subtype of **integer**) and **float**. The following types of numbers are not currently supported: **complex**, **rational** (except for its subtype **fixnum**), **ratio**, and **bignum**.

2.1.1 Integers

[*Type*]

integer

This type is a subtype of **number**. COMMON LISP defines two subtypes of **integer**: **fixnum** and **bignum**.

Compatibility note: **fixnum** is the only type of **integer** currently supported, i.e., objects of type **bignum** are not supported.

[*Type*]

fixnum

This type is a subtype of **integer**.

Implementation note: Integers in the range -2^{15} to $2^{15}-1$ (inclusive) are fixnums.

2.1.2 Ratios

Ratios are not currently supported.

2.1.3 Floating-point Numbers

[*Type*]

float

This type is a subtype of **number**. COMMON LISP defines the following subtypes of **float**: **short-float**, **single-float**, **long-float**, and **double-float**.

Implementation note: Both **single-float** and **double-float** formats are provided. **short-float** and **long-float** are equivalent to **single-float** and **double-float**, respectively.

2.1.4 Complex Numbers

Complex numbers are not currently supported.

2.2 Characters

[*Type*]

character

Objects of type **character** represent *printed glyphs*, e.g., letters (in various styles and of various alphabets and writing systems), icons, and text formatting operations. Characters have three attributes: code, bits, and font. COMMON LISP defines one subtype of **character**: **string-char**.

Implementation note: Non-zero fonts are not supported. Control and Meta bits are supported. The code attribute of a **character** conforms to the **ASCII** code.

Compatibility note: The type **character** is a subtype of **fixnum.** In other words, characters are represented by fixnums (as they are in ZETALISP).

2.2.1 Standard Characters

[*Type*]

standard-char

This type is a subtype of **string-char.** Objects of type **standard-char** make up the COMMON LISP *standard character set.* This character set is equivalent to the 95 standard **ASCII** printing characters plus a newline character. All COMMON LISP implementations must support the standard character set.

Implementation note: The *semi-standard* characters - **#\Backspace, #\Tab, #\Linefeed, #\Page, #\Return, #\Rubout** - are supported.

2.2.2 Line Divisions

In GCLISP (as in COMMON LISP), a single character, **#\Newline,** serves as a line delimiter.

Implementation note: The GCLISP interface to **PC-DOS** (or **MS-DOS**) reads an **ASCII CR/LF** pair as **#\Newline,** and writes a **#\Newline** as an **ASCII CR/LF** pair.

2.2.3 Non-standard Characters

GCLISP supports the entire **ASCII** character set (including all non-printing characters). The only **ASCII** control character given a name (besides those which are standard or semi-standard COMMON LISP characters) is **#\Escape.**

2.2.4 Character Attributes

GCLISP supports the *Control* and *Meta* bits attributes. GCLISP does not currently support non-zero font attributes.

2.2.5 String Characters

[*Type*]

string-char

Objects of this type are characters which can appear in strings, i.e., vectors of string-chars. COMMON LISP defines one subtype of **string-char: standard-char.**

2.3 Symbols

[*Type*]

symbol

Objects of this type are data structures with the following components: a print-name (also called *pname*), a property-list cell, and a package cell. (A *cell* is a component which can hold a LISP object.)

A **symbol** is usually stored in a **package**, where it can be found via its print-name.

Symbols are most commonly used as names of variables. They are also used as the names of functions, special forms, and macros.

Implementation note: Symbols have two additional components: a *value* cell and a *function* cell. These cells facilitate the symbol's role as a variable and a function name. They are used to hold the variable's current value and functional definition, respectively.

A symbol has no *print-name* cell, i.e., the print-name of a symbol is not stored as a LISP **string**. Thus, functions which return a symbol's print-name (e.g., **symbol-name**) actually create a string that is a copy of the print-name.

2.4 Lists and Conses

[*Type*]

cons

This type is a subtype of **list**. Objects of this type are data structures with two alterable components. These components have traditionally been named **car** and **cdr** (though COMMON LISP also names them **first** and **rest**). Conses are used to make singly-linked list structures, the fundamental LISP data-structures.

Note: The empty list **'()'** (i.e., the object **nil**) is *not* of type **cons** (even though it is a legal argument to the functions **car** and **cdr**). This makes sense since a cons is defined to have two *alterable* components, and the empty list has no alterable components.

[*Type*]

null

This type is a subtype of both **list** and **symbol**. There exists only one object of this type: **nil**, i.e., the empty list, **'()'**.

[*Type*]

list

The **list** is the basic data structure of LISP. A **list** is either a **cons** or the empty list, **'()'**, i.e., **nil**.

Throughout the GCLISP documentation, the term *list* refers to what COMMON LISP calls a *true list*. Thus, the phrase "must be a list" should be read as "must be a true list." A *true list* is either the empty list, or a **cons** whose **cdr** is a true list. Note that this is a recursive definition.

A *dotted-pair list* is a **list** which is not a true list, i.e., it is not terminated by **nil**.

2.5 Arrays

[*Type*]

array

Objects of this type are data structures with a user-definable number of components, which are arranged according to a *rectilinear* (i.e., Cartesian) coordinate system. The components can be accessed and updated in constant time.

One dimensional arrays, i.e., vectors, may be defined to have an additional attribute: a fill pointer.

An array which may contain elements of any type is called a *general array*. An array which has no special attributes (e.g., a fill pointer) is called a *simple array*.

Implementation note: Arrays may be defined to have an **array leader** (as in ZETALISP). An array leader functions as a simple general vector prepended to the main array. The leader is accessed and updated independently of the main array.

Compatibility note: Only vectors are currently supported. Adjustable arrays, displaced arrays, and bit-vectors are not supported. Also, array leaders are not part of COMMON LISP.

2.5.1 Vectors

[*Type*]

vector

This type is a subtype of **array**. Objects of this type are one-dimensional arrays.

The user may define the size of the vector (i.e., the number of components), the type of objects which a component may contain (e.g., **string-char**), and the existence of a fill pointer.

A vector which may contain elements of any type is called a *general vector*. A vector which has no special attributes (e.g., a fill pointer) is called a *simple vector*.

Implementation note: Two subtypes of **vector** are represented more space-efficiently than general vectors: **string** and **(vector (unsigned-byte 8))**.

Vectors may be defined to have an **array leader** (as in ZETALISP). An array leader functions as a simple general vector prepended to the main vector. The leader is accessed and updated independently of the main vector.

Compatibility note: Adjustable vectors, displaced vectors, and bit-vectors are not currently supported. Also, array leaders are not part of COMMON LISP.

2.5.2 Strings

[*Type*]

string

This type is a subtype of **vector**. More specifically, it is a specialized **vector** whose elements are of type **string-char**.

Implementation note: Strings may have an array leader (as in ZETALISP).

2.5.3 Bit-Vectors

Bit-Vectors are not currently supported.

2.6 Hash Tables

Hash Tables are not currently supported.

2.7 Readtables

A readtable defines a mapping from character objects to character types (e.g., constituent, whitespace, macro, etc.).

Implementation note: GCLISP supports a single readtable.

2.8 Packages

[*Type*]

package

A package represents a name space (i.e., a mapping from print names to symbols). All printed representations of symbols that are read by the LISP reader are mapped to their respective symbols via some package. Packages allow related symbols to be grouped apart from other symbols in order to reduce name space conflicts.

2.9 Pathnames

[*Type*]

pathname

Objects of this type are structures which are used to name files in an implementation-independent manner. Files are not LISP objects; they belong to a *file system* which is implementation dependent and external to LISP.

In spite of the differences among file systems, and hence the differences in file naming, certain attributes are common to most file systems. The components of a **pathname** correspond to these attributes. A **pathname** consists of six components: a *host*, a *device*, a *directory*, a *name*, a *type*, and a *version*.

One should think of a **pathname** as a name of a group of files (which may contain zero, one, or many actual files) which may vary over time.

Compatibility note: The **PC-DOS** (or **MS-DOS**) version of GCLISP does not currently support the *host* or *version* components.

2.10 Streams

[*Type*]

stream

Objects of this type are sources and/or sinks of data (e.g., characters, bytes, and LISP objects). **streams** serve as an implementation-independent interface to files and devices external to LISP.

Implementation note: GCLISP's streams are similar to ZETALISP's streams. For example, user-written streams are supported.

Compatibility note: User written streams are not part of COMMON LISP.

2.11 Random-States

Objects of this type are not currently supported.

2.12 Structures

[*Type*]

structure

An object of this type is a composite data structure, analogous to a record structure in Pascal. Any number of user-defined **structure** subtypes may be created, each one having its own set of constructing, accessing, and typing functions.

2.13 Functions

[*Type*]

function

An object is of type **function** if it may legally appear as the first argument to **funcall** or **apply**. **function** has the following subtypes: **compiled-function**, **closure**, **symbol**, **stream**, and **stack-group**. Also, a *lambda-expression* (a list whose first element is the symbol **lambda**) is an object of type **function**.

[*Type*]

compiled-function

This type is a subtype of **function**. An object of this type is a compiled-code object. Most of the standard GCLISP functions are **compiled-function** objects.

[*Type*]

closure

This type is a subtype of **function**. Objects of this type are functions combined with state information (as in PL/I procedures with local static variables, or Smalltalk objects with instance variables).

Compatibility note: The variables closed over by a **closure** are not shared by any other **closure**, even one defined in the same binding environment.

[*Type*]

stack-group

This type is a subtype of **function.** Objects of this type are used to represent the state of a LISP computation. They can be used to implement advanced control structures such as co-routines and generators.

Implementation note: GCLISP's stack-groups are quite similar to ZETALISP's stack groups.

Compatibility note: Stack groups are not part of COMMON LISP.

2.14 Unreadable Data Objects

The printed representation of an unreadable data object which GCLISP produces, conforms to the COMMON LISP standard.

2.15 Overlap, Inclusion, and Disjointness of Types

The data type supported by GCLISP are arranged in a subtype/supertype hierarchy that conforms to the COMMON LISP standard except for the following differences:

- In GCLISP, the type **character** is a subtype of the type **number**, while in COMMON LISP, the two types are disjoint.

- In GCLISP, the types **closure** and **compiled-function** are subtypes of the type **common**, while in COMMON LISP they are not.

- In GCLISP, the type **array** is a subtype of the type **common** even though the type **array** contains array objects with leaders, which are not of type **common.**

Chapter 3

Scope and Extent

Naming something and then referring to that thing by its name at some other place or time is a fundamental part of every language; be it a *natural* language like English, or an artificial language like COMMON LISP. Although English and COMMON LISP are very different languages, their basic concepts of naming and referring (or referencing) are quite similar.

In COMMON LISP, every *entity* can have a name. When one wants to refer to an entity, one uses its name. As in English, a name may refer to different entities at different places and times. The word *President* exemplifies the context-sensitive nature of names in English. *President* refers to a different person in different places (e.g., Gold Hill Headquarters, Washington, D.C., Paris). Within the same place, *President* may also refer to different people over the course of time. For example, within a single business meeting (held in 1984), *President* may refer to Stan Curtis, Ronald Reagan, and Francois Mitterand over the course of the the meeting.

In COMMON LISP, the region in which a name refers to a particular entity is called the the *scope* of the name. The interval of time during which a name refers to a particular entity is called the *extent* of the name. Scope concerns the spatial, textual, or *lexical* representation of a LISP form (e.g., its appearance on a piece of paper). Extent concerns the time during which the form is being evaluated.

Before a name can refer to an entity, however, a correspondence between the name and that entity must be *established*. Only functions and certain special forms (e.g., let) are able to establish names. The scope and extent of a name are relative to the form which established it. The scope of the name can be limited to or independent of the textual region which the establishing form encloses. Likewise, the extent of the name can be limited to or independent of the interval of the time during which the establishing form is being evaluated.

These various kinds of scope and extent are defined in COMMON LISP as follows:

Lexical Scope   A name which has lexical scope can only be used within the lexical (i.e., textual) region of the establishing form.

Indefinite Scope
: A name which has indefinite scope can be used anywhere, regardless of the lexical region of the establishing form.

Dynamic Extent
: A name which has dynamic extent can only be used during the interval of time between the start and finish of the evaluation of the establishing form.

Indefinite Extent
: A name which has indefinite extent can be used at any time after being established, regardless of whether the establishing form is still in the process of being evaluated.

Currently, GCLISP differs from COMMON LISP in the following way:

> The **CLRM** states that some variable names (i.e., local variable names) have *lexical scope and indefinite extent*. In GCLISP, all variable names have *indefinite scope and dynamic extent*. In other words, all variables in GCLISP are *special* variables.
>
> The **CLRM** also states that all block and tag names have *lexical scope and dynamic extent*. In GCLISP, all block and tag names have *indefinite scope and dynamic extent*.
>
> In general, wherever the **CLRM** uses the words *lexical scope*, the GCLISP user should read *indefinite scope*.

These differences will be eliminated in the near future, so the user should not write code which relies upon them.

This means that the user should use **declare** (or **defvar**, **defparameter**, etc.) to declare those variables that are intended to be special variables. GCLISP programs which use undeclared special variables will not work correctly when run on other COMMON LISP implementations.

This also means that the user should not use **go**, **return**, or **return-from** to execute a *non-local exit*; **throw** should be used instead.

For a more in-depth explanation of scope and extent, the user should read Chapter 3 of the **CLRM**.

Chapter 4

Type Specifiers

Every object in COMMON LISP is a member of at least one type. Every type in COMMON LISP has a *specifier* (i.e., a name). This chapter describes these specifiers and the functions which deal with them.

4.1 Type Specifier Symbols

The predefined data types in GCLISP are named by symbols. These symbols are listed in Table 4-1.

Table 4-1: GCLISP Standard Type Specifer Symbols

array	**integer**	**short-float**
atom	**keyword**	**single-float**
character	**list**	**stack-group**
closure	**long-float**	**standard-char**
common	**nil**	**string**
compiled-function	**null**	**string-char**
cons	**number**	**structure**
double-float	**package**	**symbol**
fixnum	**pathname**	**t**
float	**random-state**	**unsigned-byte**
function	**sequence**	**vector**

4.2 Type Specifier Lists

A type specifier may also take the form of a list. The first element of a type specifier list is always a symbol. The rest of the list provides additional type information.

4.3 Predicating Type Specifiers

Predicating type specifiers are not currently supported.

4.4 Type Specifiers that Combine

Combinatorial type specifiers are not currently supported.

4.5 Type Specifiers that Specialize

The specializing type specifier (**vector** *element-type size*) is supported. This type specifier denotes the set of one-dimensional arrays of length *size* whose elements are of type *element-type*.

element-type must be present and must be one of the following type specifiers: **t**, **string-char**, or **(unsigned-byte 8)**. *size* is optional and if present, must be a non-negative integer.

4.6 Type Specifiers that Abbreviate

Since GCLISP supports a specialized vector containing only unsigned 8-bit bytes, the following abbreviated type specifier is provided:

(unsigned-byte 8)
: Specifies the set of non-negative integers which can be represented by an 8-bit byte. This type specifier is an abbreviation for **(integer 0 255)**.

4.7 Defining New Type Specifiers

When **defstruct** is used to define a new type of structure, it also defines the name of the structure as a type specifier symbol. Currently, this is the only way of creating a new type specifier, since the **deftype** macro is not supported.

4.8 Type Conversion Function

[*Function*]

coerce *object result-type* => *result-type-object*

This function returns an object of type *result-type* that is *equivalent* to *object*.

If *object* is already of type *result-type*, *object* is simply returned unchanged. Otherwise, an equivalent object is created and returned. The following result-types are supported:

list — *object* must be a **sequence** subtype (e.g., **string**, **vector**). A list whose elements are **eql** to the elements of *object* is returned.

vector — *object* must be a **sequence** subtype (e.g., **list**, **string**). A simple general vector whose elements are **eql** to the elements of *object* is returned.

string — *object* must be a **sequence** subtype (e.g., **list**, **vector**) whose elements are all of type **character** (i.e., **characterp** is true of each element). A string whose elements are **eql** to the elements of *object* is returned.

character — *object* must be either a **string** of length 1 or a **symbol** whose print-name is of length 1. The single character which composes the string or print-name is returned.

string-char — Same effect as **character** except that the returned character contains no bits or fonts attributes.

float, short-float or **single-float**
object must be a **number.** An equivalent **single-float** number is returned.

double-float or **long-float**
object must be a **number.** An equivalent **double-float** number is returned.

t
object (which may be of any type) is simply returned.

4.9 Determining the Type of an Object

[*Function*]

type-of *object* **=>** *type-specifier*

This function returns the name of a type (i.e., a *type-specifier* to which *object* belongs.

The following type-specifiers may be returned:

closure	**pathname**
compiled-function	**single-float**
cons	**stack-group**
double-float	**symbol**
fixnum	**(vector t N)**
null	**(vector string-char N)**
package	**(vector (unsigned-byte 8) N)**

In addition, the name of a named structure (defined using **defstruct**) is the *type-specifier* for that structure.

Chapter 5

Program Structure

COMMON LISP objects are used to represent three basic conceptual (or abstract) entities: data to be manipulated, expressions to be evaluated, and functions to be applied. This chapter deals with COMMON LISP objects viewed as expressions and functions.

5.1 Forms

A *form* is a COMMON LISP object which may legally be evaluated. A more perspicuous but less accurate definition of a form is the following: A form is an expression which, when evaluated, returns a value.

Forms may be divided into the following semantic categories:

Self-Evaluating Form
: Represented by its value.

Variable
: Represented by a symbol.

Special Form
: Represented by a list whose first element is a symbol which names a special form.

Macro Call
: Represented by a list whose first element is a symbol.

Function Call
: Represented by a non-empty list.

The following pseudo-code algorithm roughly corresponds to the algorithm used by the evaluator to map objects to the above categories:

```
(cond
    ;; Self-Evaluating Forms
    ((or (numberp object) (stringp object))
     object)
    ;; Variables
    ((symbolp object)
     (symbol-value object))
```

```
((listp object)
 (let ((name (first object)))
    (cond
        ;; Special Form
        ((and (symbolp name) (special-form-p name))
            ;; Process the special form
         ...)
        ;; Macro Call
        ((and (symbolp name) (macro-function name))
            ;; Process the macro call
         ...)
        ;; Function Call
        (t
         ;; Process the function call
         ...))))
(t
 ;; Error:  Invalid form
))
```

As the pseudo-code illustrates, the only *syntactically* valid forms are numbers, strings, symbols, and lists.

The following subsections describe (conceptually) the evaluation of forms.

5.1.1 Self-Evaluating Forms

When a self-evaluating form is evaluated, the form itself is simply returned. All numbers and strings are self-evaluating forms. The symbols **t** and **nil** and all **keyword** symbols can be considered self-evaluating forms.

5.1.2 Variables

A variable is represented by a symbol. When a symbol is evaluated, the value of the variable named by the symbol is returned. In GCLISP, a symbol always names a special (dynamic) variable (See Chapter 3).

Compatibility Note: COMMON LISP specifies that a symbol can represent either a lexical or special variable, depending on the context in which it is used.

5.1.3 Special Forms

When a non-empty list is evaluated, the evaluator checks the first element of the list. If the first element is a symbol which appears in Table 5-1, then the list is a special form. Each special form is evaluated in its own particular way.

Table 5-1: GCLISP Special Form Names

and	**if**	**prog1**
block	**ifn**	**prog2**
case	**ignore-errors**	**progn**
catch	**labels**	**progv**
cond	**let**	**psetq**
condition-bind	**let***	**quote**
declare	**loop**	**return**
defun	**macro**	**return-from**
do	**multiple-value-bind**	**setq**
do*	**multiple-value-list**	**throw**
dolist	**multiple-value-prog1**	**unless**
dotimes	**multiple-value-setq**	**unwind-protect**
eval-when	**or**	**when**
function	**prog**	
go	**prog***	

Compatibility Note: The following names are defined as macros in COMMON LISP:

and	**dotimes**	**prog***
case	**loop**	**prog1**
cond	**multiple-value-bind**	**prog2**
defun	**multiple-value-list**	**psetq**
do	**multiple-value-setq**	**return**
do*	**or**	**unless**
dolist	**prog**	**when**

Currently, no equivalent macros for these special forms are provided. The following names are not defined in COMMON LISP (either as special forms or macros), they are GCLISP extensions: **condition-bind, ifn, ignore-errors, macro.**

5.1.4 Macros

If the first element of a non-empty list is a symbol which is not the name of a special form, the evaluator checks to see if the symbol has a macro definition. If the symbol is defined as a macro (e.g., via **defmacro**), the non-empty list is a macro-call form.

The evaluator applies the macro's *macro-expansion* function to the macro-call form. The result of this application is a new form. This new form is evaluated and the results are returned as the results of the original macro-call form.

5.1.5 Function Calls

If the first element of a non-empty list is neither the name of a special form nor the name of a macro, the non-empty list is a function-call form.

The evaluator assumes that the first element in the function-call form names a functional object and that the rest of the elements are forms to be evaluated (in order from left to right) to provide arguments to the functional object.

First, the evaluator evaluates each argument form, and creates an *argument* list containing the first value of each form.

Secondly, the functional object named by the first element of the function-call form is obtained. (The actual method used is logically identical to that defined by the special form **function**.)

Thirdly, this functional object is applied (see **apply**) to the argument list.

Finally, the results of the application are returned as the results of the function-call form.

Note that the above description is a logical one; the actual algorithm used to evaluate a function-call form may be quite different.

5.2 Functions

The first element in a function-call form should be a *function name*. A function name is either a symbol which has a function definition or it is a *lambda-expression*. As mentioned above, the functional object named by the function name may be obtained using the **function** special form.

5.2.1 Named Functions

A symbol can be given a function definition using the special form **defun**.

5.2.2 Lambda-Expressions

A *lambda-expression* is a human-readable description of a functional object. In other words, a lambda-expression is a *program*. A functional object is a machine executable algorithm combined with local variable binding information. When *applied* to a list of arguments, the functional object *computes* zero or more values.

Since a lambda-expression is a LISP object, and is therefore executable, it is also a functional object. In other words, the printed representation of a lambda-expression serves as a user-readable program, while the internal LISP representation of a lambda-expression serves as a functional object.

Lambda-expressions are not the only kind of functional objects; closures, compiled-functions, and stack groups are also functional objects. A functional object is *not* a form. A form is evaluated, while a functional object is *applied*.

A lambda-expression is a list which has the following syntax:

(**lambda** *lambda-list* . *body*)

The first element must be the symbol **lambda**. The symbol **lambda** does *not* name a function. Its presence at the beginning of the list is merely an indicator to procedures such as **apply** and **function** that the list is a lambda-expression.

When the functional object described by a lambda-expression is applied to a list of arguments, the following occurs (in order):

1. The lambda-list is matched against the argument list (described in more detail below).
2. The body is evaluated as an implicit **progn**.
3. The results of the implicit **progn** (i.e., the results of last form in the body) are returned as the results of the application.

The syntax of the lambda-list is as follows:

```
({Var}* [&optional {Var | (Var [initform])}*]
        [&rest Var]
        [&aux {Var | (Var [initform])}*])
```

The matching of the argument list to the lambda list is performed almost exactly as described in section 5.2.2 in the CLRM. The only differences are as follows:

- No *supplied-p* parameters (i.e., *svar* variables) are supported.
- Neither *keyword* parameters nor the &key lambda-list keyword are supported.
- All parameters are bound as special (dynamic) variables.

5.3 Top-Level Forms

The following forms are normally evaluated at Top-Level. Although the GCLISP evaluator will evaluate them correctly at locations other than Top-Level, a COMMON LISP compiler may not compile them correctly at other than Top-Level.

5.3.1 Defining Named Functions

[*Special form*]

defun *Name Lambda-list {Declaration | Doc-string}**
    *{Form}** => *name*

This special-form makes *name* the global name of the function specified by the lambda-expression

```
(lambda lambda-list
    {declaration|doc-string}*
    (block nil {form}*))
```

Compatibility note: The body of the defined function is not enclosed in a **block** construct.

5.3.2 Declaring Global Variables and Named Constants

[*Macro*]

defvar *Name* [*init-value* [*doc-string*]] => *name*

This macro is normally used at Top-Level to assign a global value to a variable. **defvar** suggests to the reader that value of the variable will be changed by the program during program execution.

name must be a symbol, which names a special variable. *init-value* must be a form. If *name* is valueless, *init-value* is evaluated and the result is assigned to *name*. Otherwise, the value of *name* is left unchanged, and *init-value* is left unevaluated.

The symbol *name* is returned.

[*Macro*]

defparameter *Name init-value* [*doc-string*] => *name*

This macro is normally used at Top-Level to assign a global value to a variable. **defparameter** suggests to the reader that the value of the variable will be set by the user before program execution in order to modify the program's behavior.

name must be a symbol, which names a special variable. *init-value* must be a form. *init-value* is evaluated and the result is assigned to *name*. The symbol *name* is returned.

(**defparameter** *name init-value*)
    <=> (**setf** *name init-value*)

[*Macro*]

defconstant *Name init-value* [*doc-string*] => *name*

This macro is normally used at Top-Level to assign a global value to a variable. **defconstant** suggests to the reader that the value of the variable will not be changed.

name must be a symbol, which names a special variable. *init-value* must be a form. *init-value* is evaluated and the result is assigned to *name*. The symbol *name* is returned.

```
(defconstant name init-value)
    <=> (setf name init-value)
```

5.3.3 Control of Time of Evaluation

eval-when is not currently supported.

Chapter 6

Predicates

A predicate is a function which tests its argument(s) for a certain property or relationship. For example, the predicate **symbolp** tests whether its argument is a symbol, while the predicate **eq** tests whether the identity relationship holds between its two arguments.

If the test succeeds, the predicate is (or returns) true; otherwise, the predicate is (or returns) false. In COMMON LISP, a predicate always returns the symbol **nil** for false and usually returns the symbol **t** for true (exceptions to the latter rule are always clearly indicated). The value name *boolean* indicates that a predicate always returns either **t** or **nil**.

6.1 Logical Values

[*Constant*]

nil => *nil*

This symbol represents two unrelated things: the logical value false, and the empty list.

The empty list may also be represented by the notation '()'. The LISP reader interprets both **nil** and '()' as referring to the constant **nil**.

nil is the only member of the type **null**, which is a subtype of both **symbol** and **list**.

[*Constant*]

t => *t*

This symbol represents the logical value true.

In COMMON LISP, the symbol **nil** represents false, while everything else, including **t**, represents **true**. Most COMMON LISP predicates return **t** to represent true, e.g., **numberp**.

6.2 Data Type Predicates

6.2.1 General Type Predicates

[*Function*]

typep *object type* => *boolean*

This function is a predicate which is true if *object* is of type *type*.

object may be an object of any type. *type* must be a type specifier.

Note that an object may be of more than one type.

Examples:

```
(typep nil 'symbol) => t
(typep nil 'list) => t
(typep 'foo 'list) => nil
```

[*Function*]

subtypep *type1 type2* => *boolean certainty*

This function is a predicate which is true if *type1* can be determined to be a subtype of *type2*.

Both *type1* and *type2* must be type specifiers.

subtypep may be false for two reasons: *type1* is not a subtype of *type2*, or the relationship between *type1* and *type2* cannot be determined. In the first case, *certainty* is **t**; in the second case *certainty* is **nil**. (If **subtypep** is true, *certainty* is always **t**.)

Examples:

```
(subtype 'null 'symbol) => t t
(subtype 'null 'cons) => nil t
```

6.2.2 Specific Data Type Predicates

[*Function*]

null *object* => *boolean*

This function is a predicate which is true if *object* is **nil** and false otherwise.

Speaking precisely, the **null** predicate is true if and only if *object* is of type **null**. The only object of type **null** is **nil**, i.e., the empty list '()'.

(null *object***)** <=> **(eq** *object* **'())**

[*Function*]

symbolp *object* => *boolean*

This function is a predicate which is true if and only if *object* is of type **symbol**.

[*Function*]

atom *object* => *boolean*

This function is a predicate which is true if and only if *object* is *not* of type **cons**. Therefore, lists (excluding the empty list) are not atoms, while everything else in COMMON

LISP (including the empty list) *is* an **atom**.

(atom *object***)** **<=>** **(not (consp** *object***))**
<=> **(or (null** *object***)**
(not (listp *object***)))**

[*Function*]

consp *object* **=>** *boolean*

This function is a predicate which is true if and only if *object* is of type **cons**. Note: **nil** is *not* of type **cons**.

(consp *object***)** **<=>** **(not (atom** *object***))**

[*Function*]

listp *object* **=>** *boolean*

This function is a predicate which is true if and only if *object* is of type **list**. An object is of type **list** if and only if it is either of type **cons** or of type **null**.

(listp *object***)** **<=>** **(or (consp** *object***)**
(null *object***))**

[*Function*]

numberp *object* **=>** *boolean*

This function is a predicate which is true if and only if *object* is some type of number, e.g. **fixnum**, **single-float**.

[*Function*]

integerp *object* **=>** *boolean*

This function is a predicate which is true if and only if *object* is of type **integer.**

Examples:

```
(integerp 1) => t
(integerp #xA) => t
(integerp 'a) => nil
(integerp #\@) => t
```

[*Function*]

floatp *object* **=>** *boolean*

This function is a predicate which is true if and only if *object* is of type **float.**

[*Function*]

characterp *object* **=>** *boolean*

This function is a predicate which is true if and only if *object* is of type **character.**

Compatibility note: character is a subtype of **integer.**

[*Function*]

stringp *object* **=>** *boolean*

This function is a predicate which is true if and only if *object* is of type **string.**

Examples:

```
(stringp "ABC") => t
(stringp 'abc) => nil
(stringp #(#\A #\B #\C)) => nil
(stringp "") => t
(stringp "a") => t
(stringp #\a) => nil
```

[*Function*]

vectorp *object* => *boolean*

This function is a predicate which is true if and only if *object* is of type **vector**, i.e., a one-dimensional **array**.

Examples:

```
(vectorp "ABC") => t
(vectorp #('foo 'bar 'baz)) => t
```

[*Function*]

arrayp *object* => *boolean*

This function is a predicate which is true if and only if *object* is of type **array**.

(arrayp *object***) <=> (vectorp** *object***)**

[*Function*]

packagep *object* => *boolean*

This function is a predicate which is true if and only if *object* is of type **package**.

[*Function*]

functionp *object* **=>** *boolean*

This function is a predicate which is true if *object* is acceptable as the first argument of **apply**, i.e., can be *applied* to a list of arguments.

```
(functionp object)
    <=> (or (symbolp object)
            (and (listp object)
                 (eq (first object) 'lambda))
            (closurep object)
            (compiled-function-p object)
            (stack-group object))
```

Examples:

```
(functionp 'car) => t
(functionp 'setq) => t
(functionp '(lambda (arg) (list arg))) => t
(functionp #'cdr) => t
(progn  (fmakunbound 'foo)
        (functionp 'foo)) => t
```

[*Function*]

compiled-function-p *object* **=>** *boolean*

This function is a predicate which is true if and only if *object* is of type **compiled-function**.

Examples:

```
(compiled-function-p #'car) => t
```

[*Function*]

closurep *object* => *boolean*

This function is a predicate which is true if and only if *object* is of type **closure.**

[*Function*]

stack-group-p *object* => *boolean*

This function is a predicate which is true if and only if *object* is of type **stack-group.**

[*Function*]

commonp *object* => *boolean*

This function is a predicate which is true if and only if *object* belongs to a type which is specified as part of the COMMON LISP core.

object may be an object of any type.

The only GCLISP data type which is not part of the COMMON LISP core is **stack-group.**

6.3 Equality Predicates

[*Function*]

eq *object1* *object2* => *boolean*

This function is a predicate which is true if and only if *object1* and *object2* are one and the same object.

Note: Two objects may look the same when printed and still be different objects.

Examples:

```
(eq (cons t t) (cons t t)) => nil
(eq (float 3) (float 3)) => nil
(eq 65. #\A) => t
(eq 7 7) => t
(eq 'foo 'foo) => t
```

[*Function*]

neq *object1 object2* => *boolean*

This function is a predicate which is true if and only if *object1* and *object2* are not one and the same object.

(neq *object1 object2***)**
  <=> (not (eq *object1 object2***))**

[*Function*]

eql *object1 object2* => *boolean*

This function is a predicate which is true if and only if *object1* and *object2* are **eq**, or they are numbers with the same type and value.

Examples:

```
(eql (cons t t) (cons t t)) => nil
(eql 65. #\A) => t
(eql (float 3) (float 3)) => t
(eql 'foo 'foo) => t
```

[*Function*]

neql *object1 object2* => *boolean*

This function is a predicate which is true if and only if *object1* and *object2* are neither **eq**, nor numbers with the same type and value.

(**neql** *object1 object2*)
    <=> (**not** (**eql** *object1 object2*))

[*Function*]

equal *object1 object2* **=>** *boolean*

This function is a predicate which is true if *object1* and *object2* are isomorphic (of identical type and structure).

Numbers and characters are **equal** if they are **eql**.

Symbols are **equal** if they are **eq**.

Conses are **equal** if their **cars** and **cdrs** are **equal**.

Arrays (other than strings) are **equal** if they are **eq**.

Strings are **equal** if they have the same length and all their characters are equal (i.e, string equality is case sensitive).

Structures are **equal** if they are of the same type and all of their components are **equal**.

In most cases, if two objects have the same printed representation, they are **equal**.

Implementation note: equal does not check for circularity in the case of structures and conses.

Examples:

```
(equal (cons t t) (cons t t)) => t
(equal (float 3) (float 3)) => t
(equal #(t t) #(t t)) => nil
(equal "abc" "abc") => t
(equal "ABC" "abc") => nil
```

6.4 Logical Operators

[*Function*]

not *object* => *boolean*

This function is a logical operator which is true if and only if *object* is the logical value false, i.e., **nil**.

This predicate may be used to logically invert a *boolean* object (i.e., **t** or **nil**).

```
(not object)     <=> (null object)
                 <=> (not (not (not object)))
```

Examples:

```
(not 0) => nil
(not nil) => t
(not "nil") => nil
```

[*Special form*]

and {*form*}* => *nil/last-form-results*

This special form serves as a logical operator and a control structure.

The forms (*form1...formn*) are evaluated, one by one, from left to right. If any *form* (e.g., *formi*) returns **nil**, **and** returns **nil** without evaluating the remaining forms (*formi+1...formn*). Otherwise, the values of the last *form* are returned.

If no arguments are provided, **and** returns **t**.

```
(and form1 form2 ... formn)
    <=> (cond   ((not form1) nil)
                ((not form2) nil)
```

```
                ...
                (t formn))
(and form) <=> form
```

Examples:

```
(and) => t
(values (and (setf foo 1)
             (incf foo)
             nil
             (incf foo))
        foo) => nil 2
(and (values nil t) t) => nil
(and t (values nil t)) => nil t
```

[*Special form*]

or {*form*}* => *non-nil-result/last-form-results*

This special form serves as a logical operator and a control structure.

The forms (*form1...formn*) are evaluated, one by one, from left to right. If any *form* (e.g., *formi*) returns a non-nil value, **or** returns that value without evaluating the remaining forms (*formi+1...formn*). Otherwise, the values of the last *form* are returned.

If no arguments are provided, **or** returns **nil**.

```
(or form1 form2 ... formn)
    <=> (cond (form1) (form2) (t formn))
(or form) <=> form
```

Examples:

```
(or) => nil
(or (< 1 2) (> 5 3)) => t
(or t nil t) => nil
(or (values nil 'foo)
    (values nil 'bar)) => nil bar
(or (values nil (setf foo 1))
    (values nil (incf  foo))
    (incf foo)
    (incf foo)) => 3
```

Chapter 7

Control Structure

GCLISP provides all of the fundamental control structures specified by COMMON LISP.

7.1 Constants and Variables

Because LISP objects are used to represent both programs and data, the special form **quote** is provided to explicitly indicate to the evaluator that an object is to be treated as a constant data object (i.e., a literal).

In COMMON LISP, variables and function names have very similar attributes:

- Both are represented by symbols.
- A variable has a value, while a function name has a function definition.
- Both may be bound (for example, variables via **let** and function names via **labels**).
- Both may be *unbound* (via **makunbound** and **fmakunbound**, respectively).

In short, function names should be viewed as just another category of variable, which may be referenced and manipulated in ways analogous to *ordinary* variables.

7.1.1 Reference

The following functions and special forms explicitly reference the values of constants, variables, and function names.

[*Special form*]

quote *Object* => *object*

This special form returns its argument unevaluated.

quote prevents the evaluator from evaluating *object* as a *form* (i.e., a LISP object which may meaningfully be evaluated).

Some forms are *self-evaluating* (i.e., they evaluate to themselves), and thus do not need to be quoted. All numbers and strings are self-evaluating.

Since **quote** is used so often, the single quote (') is predefined as a macro character equivalent of **quote**. Thus, '*object* is read as **(quote** *object***)**.

Examples:

```
(quote foo) => foo
'foo => foo
(quote (+ 2 3)) => (+ 2 3)
(car '(list 1 2)) => list
(car (list 1 2)) => 1
```

[*Special form*]

function *Function-name* => *functional-object*

This special form returns the *functional-object* named by *function-name*.

If *function-name* is a symbol, the *functional-object* (e.g., a compiled-function or a lambda expression associated with that symbol (by **defun** for example) is returned. If *function-name* is not a symbol, it is assumed to be a lambda expression and is returned unevaluated.

Since **function** is used so often, the predefined sharp-sign macro construct, **#'**, has been provided as an abbreviation. Thus **#'***function* is read as **(function** *function***)**.

Compatibility note: If *function-name* is a lambda expression, a *lexical closure* is *not* returned. Rather, **function** merely returns the lambda expression unevaluated.

Examples:

```
(function (lambda (arg) (* arg 2)))
```

```
    => (lambda (arg) (* arg 2))
(progn
    (defun foo (arg) (* arg 2))
    (function foo))
    => (lambda (arg) (* arg 2))
(mapcar #'car '((a b) (c d) (e f)))
    => (a c e)
```

[*Function*]

symbol-value *symbol* => *value*

This function returns the *value* of the variable named by *symbol*. An error is signalled if the variable is unbound.

Examples:

```
(symbol-value nil) => nil
(symbol-value (gensym)) signals an error
(symbol-value (setf foo 'bar)) => bar
```

[*Function*]

symbol-function *symbol* => *functional-object*

This function returns the *functional-object* (e.g., lambda expression, compiled-function, closure) named by *symbol*. An error is signalled if no *functional-object* is named by *symbol*.

Examples:

```
(symbol-function (gensym)) signals an error
(labels ((foo (arg) (* arg 2)))
    (symbol-function 'foo))
    => (lambda (arg) (* arg 2))
```

[*Function*]

boundp *symbol* => *boolean*

This function is a predicate which is true if and only if the variable named by *symbol* has a value.

Examples:

```
(boundp nil) => t
(boundp (gensym)) => nil
(progn (makunbound 'foo)
       (boundp 'foo)) => nil
(boundp (setf foo 'bar)) => t
```

[*Function*]

fboundp *symbol* => *boolean*

This function is a predicate which is true if and only if *symbol* has a function definition, e.g., a lambda expression, a compiled-function, a macro.

Examples:

```
(fboundp 'car) => t
(fboundp (gensym)) => nil
(progn (defun foo (arg) (* arg 2))
       (fboundp 'foo)) => t
(progn (fmakunbound 'foo)
       (fboundp 'foo)) => nil
```

[*Function*]

special-form-p *symbol* => *nil/special-form-function*

This function is a predicate which is true if and only if *symbol* is the name of a special-form.

Instead of returning **t** for true, **special-form-p** returns a function that can interpret a special form whose name is *symbol*. When this function is applied to the rest of the special form, the effect is identical to evaluating the whole special form.

```
(apply (special-form-p 'special-form-name)
       'body) <=> (special-form-name . body)
```

Examples:

```
(special-form-p 'quote)
    => #<compiled-function ????:????>
(special-form-p 'car) => nil
(special-form-p 'do)
    => #<compiled-function ????:????>
```

7.1.2 Assignment

The following functions, macros, and special forms alter the current value of a variable.

[*Special form*]

setq *{Symbol form}** => *last-form-result*

This special form is the *simple variable assignment statement* of LISP. Each *symbol* names a variable. The value of each form is assigned to the variable which precedes it.

The assignments are performed *sequentially*, i.e., the *n*th assignment is performed before the *n*th+1 *form* is evaluated. The value of the last *form* is returned (multiple values are not passed back).

If no *symbol/form* pairs are supplied, **nil** is returned.

Examples:

```
(setq) => nil
(progn (setq foo (+ 2 3))
    (symbol-value 'foo)) => 5
(setq foo (values 2 3)) => 2
(progn (setq foo 1 foo 2) foo) => 2
(setq foo 1 bar (+ foo 1) baz (+ bar 1)) => 3
```

[*Special form*]

psetq {*Symbol form*}* **=>** *nil*

This special form is the *simple (parallel) variable assignment statement* **of LISP. It is similar to setq except that the assignments are performed** *in parallel*, **i.e., each** *form* **is evaluated (in order from left to right), then the variables are assigned the resulting values (in order from left to right).**

nil is always returned.

Examples:

```
(psetq) => nil
(values (psetq foo 666) foo) => nil 666
(progn (psetq foo 1 foo 2) foo) => 2
(progn (setq foo 1)
       (psetq foo 2 bar (+ foo 2))
      bar) => 3
```

[*Function*]

set *symbol form* **=>** *form-result*

This function assigns the value of *form* **to the variable named by** *symbol*. **set is similar to setq, except that the former evaluates its first argument and the latter does not.**

Examples:

```
(progn (set (car '(foo bar)) 5) foo) => 5
(progn (setq foo 'bar)
       (set foo 5)
       (values foo bar)) => bar 5
```

[*Function*]

makunbound *symbol* => *symbol*

This function causes the variable named by *symbol* to have no current value. A better name for this function might be **makvalueless**, since it actually does not undo the current binding; rather it leaves the variable bound (if it already is so) but valueless.

Examples:

```
(values (setf foo 'value)
        (let ((foo))
            (boundp (makunbound foo)))
        foo) => value nil value
```

[*Function*]

fmakunbound *symbol* => *symbol*

This function causes *symbol* to have no current function definition. A better name for this function might be **fmakvalueless**, since it actually does not undo the current binding; rather it leaves the function name bound (if it already is so) but undefined.

Examples:

```
(values (fboundp '1-)
        (labels ((1- (int) (- 1 int))
            (fmakunbound '1-)
            (fboundp '1-))
        (fboundp '1-)) => t nil t
```

7.2 Generalized Variables

A generalized variable is (not surprisingly) a generalization of the concept of an ordinary variable. Conceptually, an ordinary variable is defined in terms of three entities: a data structure with a *value* component (i.e., a symbol), an *access* form (i.e., the symbol itself, or the function-call form **(symbol-value** *object***)**), and an *update* form (i.e., **(setq** *object new-value***)**).

Analogously, a generalized variable is defined in terms of three entities: a data structure, which consists of one or more components; an access form, which obtains the value stored in one of the data structure's components; and an update form, which stores a new value in that same component (and returns the new value). (Actually, GCLISP further generalizes this concept by allowing multiple-value generalized variables.)

For example, the access form **(car** *cons***)** *names* a generalized variable. The data structure is a cons, the access form is the name of the generalized variable, and the form **(rplaca** *cons new-value***)** is the update form. (Actually, this is somewhat inaccurate, since **rplaca** does not return the new value.)

The **setf** macro, given the name of a generalized variable (i.e., an access form) and a new value, returns the appropriate update form. Therefore, the user no longer needs to remember any update or assignment forms. A general rule of thumb is, "If the user can access it, **setf** can update it." Thus, **setf** makes most update and assignment procedures (e.g., **setq**, **set**, **rplaca**, etc.) obsolete. **setf** supports the following access functions:

aref	**nthcdr**
array-leader	**rest**
c...r **(e.g, car, cdaar)**	**second**
fill-pointer	**symbol-function**
first	**symbol-plist**
get	**symbol-value**
getf	**third**
nth	**values**

In addition, the access function defined by **defstruct** can be used with **setf**.

The user is strongly encouraged to use **setf** for any kind of assignment.

The following generalized variable macros are also implemented by GCLISP: **getf**, **remf**, **incf**, **decf**, **push**, and **pop**.

[*Macro*]

setf {*place new-value*}* **=>** *last-new-value-form-result*

This macro produces a form which, when evaluated, updates the value at *place* to *new-value*.

place must be a form (e.g., a variable, a function-call) which, when evaluated, accesses some LISP object. *new-value* may be any form whose value may legally be assigned to the location designated by *place*.

Most of the *access* functions predefined by COMMON LISP can be handled by **setf**. These functions include: **car** and **cdr** and all their combined forms (e.g., **cadr**, **cddr**, etc.), **nth**, **get**, **aref**, **symbol-value**, and **symbol-function**. Also, access functions defined by **defstruct** can be used with **setf**.

Multiple *place/new-value* pairs are processed sequentially. **setf** returns the value of the last *new-value*. If given no arguments, it returns **nil**.

Compatibility note: If *place* is a **getf** form, **setf** may not return the value of *new-value*. Also, subforms of *place* may be evaluated more than once.

Examples:

```
(let ((foo '(a b c)))
     (setf (cadr foo) 2) foo) => (a 2 c)
(let ((foo '#(a b c)))
     (setf (aref foo 3) (+ 2 3))
     (aref foo 3)) => 5
```

7.2.1 Defining New Generalized Variables

GCLISP allows the user to define new generalized variables in a straight-forward manner. However, the methods used are different from those specified by COMMON LISP and are likely to change.

Two basic methods are provided. The first method defines the new generalized variable by a mapping from its access form to an *expanded* access form composed of one or more access forms that are already known to **setf**. The second method defines the new generalized variable by a mapping from its access form and a new-value form to a corresponding update form.

Both types of mappings are properties of the symbol that names the access form (e.g., the symbol **cddr**). The value of the property **setf-expander** is a mapping from an access form to an expanded access form. The value of the property **setf** is a mapping from an access form and a new-value form to an update form.

Both types of mappings can be defined using the concept of a *template*. In this context, a template refers to a form in which all argument forms are represented by variables. For example, a template which corresponds to the form (car '(1 2 3)) is (**car foo**).

If the value of the **setf-expander** property is a dotted-pair (i.e., a cons), the **first** element is a template of the access form being defined and the **rest** of the list is a template of an equivalent, expanded access form. For example,

```
(get 'cddr 'setf-expander)
    => ((cddr list) . (cdr (cdr list)))
```

If the value of the **setf-expander** property is not a dotted-pair, it must be a symbol which names a function of one argument. The function is called with the given access form as its argument. The function should return an expanded access form.

The value of the **setf** property is handled in an analogous manner. If the value of the property is a dotted-pair, the **first** element is a template of the access form and the **rest** of the list is a template of the appropriate update form. For example,

```
(get 'symbol-value 'setf)
    => ((symbol-value symbol) . (set symbol val))
```

Note that in the update template, the new-value argument form must be represented by the variable **val**.

If the value of the **setf** property is not a dotted-pair, it must be a symbol which names a function of two arguments. The function is called with the given access form and new-value form (in that order). It should return an update form.

Implementation note: setf uses the following internal functions: **aref-setf, aset, fset, putprop, putf, setplist, values-setf.**

7.3 Function Invocation

[*Function*]

apply *function arg* **&rest** *more-args*
    => *function-application-results*

This function applies *function* to a list of arguments and returns the results of this functional application.

function may be a lambda expression, a closure, a compiled-function, a stack-group or a symbol. If *function* is a symbol, its function definition may not be a macro or special form definition.

Conceptually, the argument list which *function* is applied to is constructed by applying **list*** to the arguments following *function*. The last argument to **apply** must be a list.

```
(apply #'fn (list arg1 ... argn))
    <=> (funcall #'fn arg1 ... argn)
```

Examples:

```
(apply #'+ '(1 2)) => 3
(apply '- 10 1 (list 5 2)) => 2
(apply #'values '(1 2 3 4)) => 1 2 3 4
(apply #'list '()) => nil
```

[*Function*]

funcall *function* **&rest** *arguments* => *function-call-results*

This function calls *function* with *arguments* and returns the results of this function call.

function may be a lambda expression, a closure, a compiled-function, a stack-group or a symbol. If *function* is a symbol, its function definition may not be a macro or special form definition.

```
(funcall #'fn arg1 ... argn)
    <=> (apply #'fn (list arg1 ... argn))
```

Examples:

```
(funcall #'+ 1 2) => 3
(funcall '- 3 2) => 1
```

```
(funcall #'+) => 0
(funcall #'values 1 2 3 4) => 1 2 3 4
```

7.4 Simple Sequencing

[*Special form*]

progn *{form}** => *last-form-results*

This special form evaluates each *form*, in order from left to right. It returns the values returned by the last *form*; the results of the other forms are simply discarded.

Note that one of the forms may cause control to be transferred to outside the **progn** (e.g., **throw**, **error**). In this case, the remaining forms are not evaluated.

Examples:

```
(progn) => nil
(progn (setf a 1)
       (setf b 2)
       (values a b)) => 1 2
```

[*Special form*]

prog1 *first-form {form}** => *first-form-result*

This special form evaluates each *form*, in order from left to right. It returns the value returned by *first-form*; the results of the other forms are simply discarded.

prog1 always returns a single value, even if *first-form* returns multiple values.

```
(prog1 a1 a2 ... an)
    <=> (let ((val a1)) a2 ... an val)
```

Examples:

```
(progn (setf foo 1)
       (prog1 foo
              (setf foo (+ foo 1)))) => 1
```

[*Special form*]

prog2 *first-form second-form {form}** => *second-form-result*

This special form evaluates each *form*, in order from left to right. It returns the value returned by *second-form*; the results of the other forms are simply discarded.

prog2 always returns a single value, even if *second-form* returns multiple values.

(prog2 *a1 a2 a3 ... an***)**
    <=> (progn *a1* **(prog1** *a2 a3 ... an***))**

Examples:

```
(progn (setf foo 1)
       (prog2 (setf foo (+ foo 1))
              foo
              (setf foo (+ foo 1)))) => 2
```

7.5 Establishing New Variable Bindings

[*Special form*]

let ({*Var* | (*Var value*)}*)
    {*declaration*}* {*form*}* => *last-form-results*

This special form establishes a binding of each specified variable to its respective value. All bindings are dynamic (i.e., of indefinite scope and dynamic extent).

Each variable is specified by a *var*, which is a symbol that names the variable. Each variable which occurs in a *(var* *value)* pair is bound to the value returned by the evaluation of the form *value*. Each *var* that occurs alone is bound to the object **nil**.

All of the *value* forms are evaluated (in order from left to right) before any of the bindings are established. Then, each of the bindings is established in an undefined order. Once all the bindings have been established, each *form* is then evaluated, in order from left to right, and the values of the last form are returned (i.e., the body of the **let** is an implicit **progn**).

Examples:

```
(let (a (b) (c nil) (d ()) (e '()))
    (values a b c d e))
    => nil nil nil nil nil
(let ((a 1) (b 2) (c 3))
    (values a b c)) => 1 2 3
(let ((a 1))
    (list (let ((a 10) (b (incf a)))
             (list a b))
          a)) => ((10 2) 2)
```

[*Special form*]

let* ({*Var* | (*Var value*)}*)
    {*declaration*}* {*form*}* => *last-form-results*

This special form establishes a binding of each specified variable to its respective value. All bindings are dynamic (i.e., of indefinite scope and dynamic extent).

let* is identical to **let** except that the variables are bound in sequence, i.e., *valuei* is evaluated and bound to *vari* before *valuei+1* is evaluated and bound to *vari+1*.

Examples:

```
(let (a (b (cons 1 a)) (c (cons 2 b)))
    (values a b c)) => nil (1) (2 1)
(let ((a 1) (b (incf a)) (c (incf b)))
    (list a b c)) => (2 3 3)
```

[*Special form*]

progv *symbol-list value-list* {*form*}* => *last-form-results*

This special form establishes a binding of each specified variable to its respective value. All bindings are dynamic (i.e., of indefinite scope and dynamic extent).

The specified variables are named by symbols which are members of the list that is the result of evaluating the form *symbol-list*. The variables' respective values are the members of the list that is the result of evaluating the form *value-list*. In other words, the variable named by the *n*th symbol in *symbol-list* is bound to the *n*th value in *value-list*. The order in which the bindings are established is undefined.

Once all the bindings have been established, each *form* is then evaluated, in order from left to right, and the values of the last form are returned (i.e., the body of the **progv** is an implicit **progn**).

Examples:

```
(progv (list a b c) (list 1 2 3)
    (values a b c)) => 1 2 3
```

[*Special form*]

labels ({(*Name Lambda-list*
{*Declaration* | *Doc-string*}* {*Form*}*)}*) {*form*}*
=> *last-form-results*

This special form establishes locally named functions.

7.6 Conditionals

[*Special form*]

if *test then* [*else*] => *last-evaled-form-results*

This special form evaluates either the *then* form or the *else* form depending on the value of the *test* form.

If the result of evaluating the *test* form is non-**nil**, the *then* form is evaluated and its results are returned by **if**.

Otherwise, if the result of evaluating the *test* form is **nil**, the *else* form is evaluated and its results are returned by **if**. If there is no *else* form, **nil** is returned.

```
(if test then else)
    <=> (cond (test then) (t else))
(if test then) <=> (if test then nil)
```

Examples:

```
(if t 1 2) => 1
(if nil 1 2) => 2
(if (not t) t) => nil
(if (setf foo 1)
    (incf foo)
    (decf foo)) => 2
```

[*Special form*]

ifn *test then* [*else*] => *last-evaled-form-results*

This special form evaluates either the *then* form or the *else* form depending on the value of (**not** *test*).

```
(ifn test then else)
    <=> (if (not test) then else)
```

Examples:

```
(ifn t 1 2) => 2
(ifn nil 1 2) => 1
(ifn (not nil) t) => nil
(ifn (setf foo 1)
     (incf foo)
     (incf foo)) => 2
```

[*Special form*]

when *test {form}** **=>** *nil/last-form-results*

If the result of evaluating *test* is non-**nil**, this special form evaluates each *form* and returns the results of the last *form*; otherwise the forms are not evaluated and **nil** is returned.

```
(when test f1 ... fn)
    <=> (cond (test f1 ... fn))
    <=> (and test (progn f1 ... fn))
    <=> (if test (progn f1 ... fn) nil)
```

Examples:

```
(when t (values 1 2 3)) => 1 2 3
(when (not t) (values 1 2 3)) => nil
```

[*Special form*]

unless *test {form}** **=>** *nil/last-form-results*

If the result of evaluating *test* is **nil**, this special form evaluates each *form* and returns the results of the last *form*; otherwise the forms are not evaluated and **nil** is returned.

```
(unless test f1 ... fn)
    <=> (cond ((not test) f1 ... fn))
    <=> (and (not test) (progn f1 ... fn))
    <=> (if test nil (progn f1 ... fn))
```

Examples:

```
(unless t (values 1 2 3)) => nil
(unless (not t) (values 1 2 3)) => 1 2 3
```

[*Special form*]

cond {(*test* {*form*}*)}* => *nil/last-evaled-form-results*

This special form is the basic conditional form of COMMON LISP. It is analogous to the if-then-elsif statement of other languages. Each (*test form1 ... formn*) component is called a *clause*. The clauses are tested sequentially, in order from left to right. The first clause that has a *test* which evaluates to a non-nil result is selected. None of the subsequent clauses are tested.

form1 ... formn of the selected clause are evaluated, and the results of *formn* are returned as the result of **cond** (i.e., *form1 ... formn* constitute an implicit **progn**). If no forms follow the *test* in the selected clause, the single value of the *test* is returned as the result of **cond**.

If no clause is selected (i.e., no *test* is true), **cond** returns **nil**.

Examples:

```
(cond) => nil
(let ((x 1)) (cond ((> x 0) 'positive)
                   ((< x 0) 'negative)
                   (t 'zero))) => positive
```

[*Special form*]

case *keyform* {({({*Key*}*) | *Key*} {*form*}*)}*
    => *nil/last-evaled-form-results*

This special form is a conditional control structure that selects at most one of its clauses, the selection being based on a key.

A clause has the following structure:

(*key-spec form1 ... formn*)

The *key-spec* must be one of the following: a list of keys (which may be objects of any type), the symbols **t** or **otherwise**, or a single *key* (which cannot be a list or the symbols **t** or **otherwise**). The symbols **t** and **otherwise** may only appear in the last clause. Duplicate keys are not allowed.

First, *keyform* is evaluated to produce a *selector-key*. Then the *key-spec* of each clause is *tested* against the *selector-key*. A *key-spec* satisfies the test if *key-spec* is a list of keys and **(member** *selector-key key-spec***)** is true; or *key-spec* is a single *key* and **(eql** *selector-key key-spec***)** is true; or *key-spec* is either the symbol **t** or the symbol **otherwise**.

The order in which clauses are checked is undefined; except that if a clause with **t** or **otherwise** as a *key-spec* occurs, it is checked last.

The first clause that contains a *key-spec* which satisfies the test is selected: its forms are evaluated and the results of the last *form* are returned, i.e., the forms are evaluated as an implicit progn. A clause containing no forms (other than the *key-spec*), returns **nil**.

If no clause is selected, **case** returns **nil**.

Examples:

```
(case t
    ((t nil) (values 1 t))
    (t (values 2 nil))) => 1 t
(case 1) => nil
(case (+ 1 1) ((2 4 6 8 10) 'even)
              ((1 3 5 7 9) 'odd)
              (otherwise '>10)) => even
```

7.7 Blocks and Exits

[*Special form*]

block *Name* {*form*}* **=>** *last-form-results*

This special form establishes *name* as the name of the **block** and then evaluates each *form* in order from left to right.

If **return-from** form that specifies *name* is evaluated within the extent of the **block**, **block** immediately returns the results specified in the **return** or **return-from** form. (If *name* is **nil**, **return** may be used instead of **return-from**.) Otherwise, **block** returns the results of the last *form*.

Compatibility note: The *name* established by **block** has dynamic scope.

[*Special form*]

return-from *Name result*

This special form causes the most recently established **block** form named *name* to be immediately returned from, returning the values of *result*.

Implementation note: An error occurs if a **(return-from** *name* **...)** is attempted outside the scope or extent of the **block** named *name*.

Compatibility note: The scope of a *name* is dynamic.

Examples:

```
(block foo
    (return-from foo (values 1 2 3))
    (values "Never Happens")) => 1 2 3
```

[*Special form*]

return *result*

This special form causes the most recently established **block** form named **nil** (e.g. **do, prog**) to be immediately returned from, returning the values of *result*.

(return *form***) <=> (return-from nil** *form***)**

7.8 Iteration

7.8.1 Indefinite Iteration

[*Special form*]

loop {*form*}*

This special form repeatedly evaluates *form1* ... *formn* in order from left to right until some form which exits the loop (e.g., **throw**) is evaluated.

loop establishes an implicit **block** named **nil**, so a **return** will exit the loop.

7.8.2 General Iteration

[*Macro*]

do ({(*Var* [*init* [*step*]])}*)
    (*end-test* {*end-form*}*) {*declaration*}*
    {*Tag* | *statement*}*
    => *nil/last-endform-results*

This macro is a general purpose iteration control structure. It consists of three parts:

An index-spec:

((*var1 init1 step1*) ...
 (*varn initn stepn*))

An end-spec: (*end-test end-form1* ... *end-formn*)

A body: *tagbody*

First, **do** establishes a binding for each of the *index* variables named by the symbols, *var1* ... *varn*. Each variable is bound to the value of its respective *init* form, or to **nil** if it has no associated *init* form. This binding is performed in parallel as in a **let** form.

Then, iteration begins. One iterative cycle consists of the following steps:

1. The *end-test* form is evaluated. If the result is non-nil, the end-forms are evaluated in order as an implicit progn, and the results of the last form are returned. If there are no end-forms, **nil** is returned. (Note that the end-spec has the same syntax as a cond clause.)

2. Otherwise, if the value of *end-test* is **nil**, the forms following the end-spec are evaluated as an implicit *tagbody*.

3. When the end of the *tagbody* is reached, each index variable is *stepped* i.e., assigned the value of its respective *step* form. This is done in parallel as in a **psetq** form. An index variable without an associated *step* form is not stepped. Then the cycle is repeated, beginning with step one.

The entire **do** control structure is executed within an implicit **block** named **nil**; thus, **do** may be exited at any point by executing the **return** form.

Examples:

```
(do ((a 1 (+ 1 a)) (b 0))
    ((= a 11) b)
    (setf b (+ b a))) => 55
```

[*Macro*]

do* ({(*Var* [*init* [*step*]])}*)
(*end-test* {*end-form*}*) {*declaration*}*
{*Tag* | *statement*}*
=> *nil/last-endform-results*

This macro is a general purpose iteration control structure. It is identical to the **do** macro except that the index variables are bound sequentially using **let***. (**do** binds its index variables in parallel using **let**.)

Examples:

```
(do* ((a 1 (+ 1 a)) (b a (+ b a)))
     ((= a 10) b)) => 55
```

7.8.3 Simple Iteration Constructs

[*Macro*]

dolist (*Var listform* [*resultform*])
    {*declaration*}* {*Tag* | *statement*}*
    => *resultform-results*

This macro provides simple iteration over the elements of a list. The *body* of this form, i.e., the sequence of tags and statements, is an implicit *tagbody*.

First, the *listform* is evaluated and must produce a list. Then, for each element in that list, the variable named by the symbol *var* is bound to that element and the body is evaluated.

Finally, the *resultform* (which if not provided, defaults to **nil**) is evaluated and its values are returned. The variable named by *var* is bound to **nil** during the evaluation of *resultform*. The **dolist** form may be exited at any time by evaluating the **return** form. This is because the **dolist** form is implicitly wrapped in a **block** named **nil**.

Examples:

```
(let ((foo '(a b c d)) (bar '()))
    (dolist (ele foo bar)
        (setf bar (cons ele bar))))
    => (d c b a)
```

[*Macro*]

dotimes (*Var countform* [*resultform*])
    {*declaration*}* {*Tag* | *statement*}*
    => *resultform-results*

This macro provides simple iteration over a sequence of integers. The *body* of this form, i.e., the sequence of tags and statements, is an implicit *tagbody*.

First, the *countform* is evaluated and must produce an integer (call it *count*). Then, for each integer in the range 0 (inclusive) to *count* (exclusive), the variable named by the symbol *var* is bound to that integer and the body is evaluated. If *count* is less than one, the body is not evaluated.

Finally, the *resultform* (which if not provided, defaults to **nil**) is evaluated and its values are returned. The variable named by *var* is bound to *count* when *resultform* is evaluated.

The **dotimes** form may be exited at any time by evaluating the **return** form. This is because the **dotimes** form is implicitly wrapped in a **block** named **nil**.

Examples:

```
(let (foo)
    (dotimes (cnt 3 foo)
        (setf foo (cons cnt foo))))
    => (2 1 0)
```

7.8.4 Mapping

[*Function*]

mapcar *function list* **&rest** *more-lists* => *result-list*

This function successively applies the functional object, *function*, to the *n* successive elements in each of the lists following *function* (where *n* is the length of the shortest *list*).

In other words, *function* is applied to the **first** element of each *list*, then the **second** element of each *list*, and so on until no more elements remain in one of the lists.

mapcar returns the *list* of the successive results of applying *function*. The first argument to **mapcar** must be a functional object which is acceptable to **apply** and which takes as many arguments as there are remaining arguments (which must all be lists).

Examples:

```
(mapcar #'list '(a b c)) => ((a) (b) (c))
(mapcar #'list '(a b c) '(1 2))
    => ((a 1) (b 2))
```

```
(mapcar #'+ '(1 2 3) '(4 5 6)) => (5 7 9)
```

[*Function*]

maplist *function list* **&rest** *more-lists* => *result-list*

This function successively applies the functional object, *function*, to the *n* successive sublists of each of the lists following *function* (where *n* is the length of the shortest *list*).

In other words, *function* is applied to all the lists, then to the **cdr** of each list, then to the **cddr** of each list, and so on until one of the sublists is **nil**.

maplist returns the list of the successive results of applying *function*.

The first argument to **maplist** must be a functional object which is acceptable to **apply** and which takes as many arguments as there are remaining arguments (which must all be lists).

Examples:

```
(maplist #'(lambda (x) x)
         '(a b c)) => ((a b c) (b c) (c))
(maplist #'append '(a b c) '(1 2 3))
    => ((a b c 1 2 3) (b c 2 3) (c 3))
(let ((foo '(1 2 7 4 6 5)))
    (maplist #'(lambda (xl yl)
                      (< (car xl) (car yl)))
             foo (cdr foo)))
    => (t t nil t nil)
```

[*Function*]

mapc *function list* **&rest** *more-lists* => *list*

This function successively applies the functional object, *function*, to the *n* successive elements in each of the lists following *function* (where *n* is the length of the shortest *list*).

In other words, *function* is applied to the *first* element of each *list*, then the *second* element of each *list*, and so on until no more elements remain in one of the lists.

mapc returns its second argument.

The first argument to **mapc** must be a functional object which is acceptable to **apply** and which takes as many arguments as there are remaining arguments (which must all be lists).

```
(mapc function list1 ... listn) <=>
    (prog1 list1
       (mapcar function list1 ... listn))
```

Examples:

```
(let ((foo '(1 2 3 4 5)) (bar 0))
    (mapc #'(lambda (x) (setf bar (+ bar x)))
          foo) bar) => 15
```

[*Function*]

mapl *function list* **&rest** *more-lists* => *list*

This function successively applies the functional object, *function*, to the *n* successive sublists of each of the lists following *function* (where *n* is the length of the shortest *list*).

In other words, *function* is applied to all the lists, then to the **cdr** of each list, then to the **cddr** of each list, and so on until one of the sublists is **nil**.

mapl returns its second argument.

The first argument to **mapl** must be a functional object which is acceptable to **apply** and which takes as many arguments as there are remaining arguments (which must all be lists).

```
(mapl function list1 ... listn) <=>
    (prog1 list1
       (maplist function list1 ... listn))
```

Examples:

```
(mapl #'(lambda (x)
          (unless (null (rest x))
          (setf (second x)
                (+ (first x)
                   (second x)))))
      '(1 2 3 4 5)) => (1 3 6 10 15)
```

[*Function*]

mapcan *function list* **&rest** *more-lists* => *result-list*

This function successively applies the functional object, *function*, to the *n* successive elements in each of the lists following *function* (where *n* is the length of the shortest *list*).

In other words, *function* is applied to the **first** element of each *list*, then the **second** element of each *list*, and so on until no more elements remain in one of the lists. **mapcan** returns the **nconc** of the successive results of applying *function*.

The first argument to **mapcan** must be a functional object which is acceptable to **apply** and which takes as many arguments as there are remaining arguments (which must all be lists).

(mapcan *function list1 ... listn***)** **<=>**
    (apply #'nconc
        (mapcar *function list1 ... listn***))**

Examples:

```
(mapcan #'(lambda (l e)
            (when (member e l) (list l)))
    '((1 2 3) (a b c) (8 9 0) (k o p) (y u 9))
    '(2 c 1 p v)) => ((1 2 3) (a b c) (k o p))
```

[*Function*]

mapcon *function list* **&rest** *more-lists* => *result-list*

This function successively applies the functional object, *function*, to the *n* successive sublists of each of the lists following *function* (where *n* is the length of the shortest *list*).

In other words, *function* is applied to all the lists, then to the **cdr** of each list, then to the **cddr** of each list, and so on until one of the sublists is **nil**.

mapcon returns the **nconc** of the successive results of applying *function*.

The first argument to **mapcon** must be a functional object which is acceptable to **apply** and which takes as many arguments as there are remaining arguments (which must all be lists).

```
(mapcon function list1 ... listn) <=>
    (apply #'nconc
        (maplist function list1 ... listn))
```

Examples:

```
(mapcon #'(lambda (x)
            (if (null (rest x))
                (list (first x))
                (list (first x) 'and)))
        '(a b c d e))
    => (a and b and c and d and e)
```

7.8.5 The "Program Feature"

[*Special form*]

prog ({*Var* | (*Var* [*init*])}*) {*declaration*}*
    {*Tag* | *statement*}* => *nil*

This special form provides the binding environment of the **let** form with the ability to perform both structured and unstructured control transfer via the **return** and **go** forms, respectively.

prog consists of two parts: a binding-spec, specified by the first component; and a body, specified by the remaining components.

The binding-spec establishes a binding environment exactly as **let** does.

The body consists of tags, which must be symbols or integers, and statements, which must be lists. The items in the body are normally processed from left to right. Tags are not evaluated, but statements are. If the end of the body is reached, **nil** is returned.

A **go** form may be evaluated in order to transfer control to a specified *tag*. A **return** form, may be used to exit from the **prog** before the end of the body is reached.

Compatibility note: Tags are dynamically scoped.

Examples:

```
(prog ((arg 5)) (+ arg 1)) => nil
(prog ((arg 5)) (return-from nil (+ arg 1)))
    => 6
(prog () (return 1 2 3)) => 1 2 3
```

[*Special form*]

prog* ({*Var* | (*Var* [*init*])}*) {*declaration*}*
    {*Tag* | *statement*}* => *nil*

This special form provides the binding environment of the **let*** form with the ability to perform both structured and unstructured control transfer via the **return** and **go** forms, respectively.

It is identical to the **prog** form, except that **let*** is used instead of **let**.

Examples:

```
(prog* ((foo 1) (bar (+ foo 1)))
    (return foo bar)) => 1 2
```

[*Special form*]

go *Tag*

This special form is used to transfer control within a *tagbody*, e.g., within a **prog** special form.

When the go form is evaluated, control is transferred to the point following *tag*. *tag* must be a symbol or an integer.

Compatibility note: Tags have indefinite scope and dynamic extent. Therefore one can go to a *tag* in a tagbody from a place within the dynamic extent of the tagbody, and yet not within the lexical scope of that tagbody. This feature should not be relied upon, since it will change in the future.

Examples:

```
(prog ()
    (go skip)
    (return 1)
    skip
    (return 2)) => 2
```

7.9 Multiple Values

7.9.1 Constructs for Handling Multiple Values

[*Function*]

values *argument* **&rest** *arguments* **=>** *results*

This function returns *n* values when given *n* arguments.

Compatibility note: values requires at least one arg., i.e., zero values cannot be returned.

Examples:

```
(values 1 2 3) => 1 2 3
(values (values 1 2) (values 3 4)) => 1 3
```

[*Function*]

values-list *list* => *list-elements*

This function returns, as multiple values, all the elements of *list*.

(values-list *list*) **<=> (apply #'values (or** *list* **'(nil))**

Compatibility note: If *list* is the empty list, i.e., **nil**, **values-list** returns a single argument, **nil**.

Examples:

```
(values-list nil) => nil
(values-list '(nil)) => nil
(values-list '(1 2 3)) => 1 2 3
```

[*Special form*]

multiple-value-list *form* => *results-list*

This special form returns a list containing the values returned by *form*.

Examples:

```
(multiple-value-list (+ 2 2)) => (4)
    (multiple-value-list (values 1 2 3))
    => (1 2 3)
```

[*Special form*]

multiple-value-prog1 *first-form* {*form*}*
    => *first-form-results*

This special form evaluates its argument forms in order from left to right. It returns the values returned by *first-form*; the results of the other forms are simply discarded.

multiple-value-prog1 returns multiple values if *first-form* returns multiple values. Thus, it is exactly like **prog1**, except that **prog1** always returns only a single value.

```
(multiple-value-prog1 f1 f2 ... fn) <=>
    (let ((vlf1 (multiple-value-list f1)))
        f2 ... fn (values-list vlf1))
```

Examples:

```
(multiple-value-prog1 (values 1 2 3)) => 1 2 3
(let ((foo 2))
    (multiple-value-prog1
        (values foo (* foo foo))
        (setf foo (+ foo 1)))) => 2 4
```

[*Special form*]

multiple-value-bind ({*Var*}*) *values-form*
    {*declaration*}* {*form*}*
    => *last-form-results*

This special form establishes a binding for each specified variable and then evaluates its body as an implicit **progn**.

First, the *values-form* is evaluated and then the variables named by the *var* symbols are bound to these values (the *i*th *var* is bound to the *i*th value). If there are more variables than values, the excess variables are bound to **nil**. If there are more values than variables, the excess values are discarded. The remaining forms are then executed and the values of the last form are returned, i.e., the body is an implicit **progn**.

[*Special form*]

multiple-value-setq ({*Var*}*) *form* => *form-result*

This special form assigns the specified variables to the values returned by *form*.

The variables named by the *var* symbols are assigned (not bound) to the values returned by *form*, repectively. If there are more variables than values, the excess variables are bound to **nil**. If there are more values than variables, the excess values are discarded.

The first value returned by *form* is returned.

Examples:

```
(let (a b c)
    (multiple-value-setq (a b c)
                         (values 1 2 3 4))
    (values a b c)) => 1 2 3
(multiple-value-setq nil (values 1 2)) => 1
```

7.9.2 Rules Governing the Passing of Multiple Values

GCLISP adheres to the COMMON LISP rules governing the passing of multiple values.

7.10 Dynamic Non-local Exits

[*Special form*]

catch *tag* {*form*}* => *throw-results/last-form-results*

This special form provides a control structure (called a *catcher*) which allows non-local, dynamically scoped exits via the evaluation of a **throw** form.

First, the *tag* form is evaluated, and its value is used as the tag of the catcher. Then the remaining forms are evaluated in order from left to right and the values of the last *form* are returned, i.e., the forms are an implicit **progn**.

However, if the evaluation of a **throw** form produces a *thrower* whose tag is **eq** to the tag of the catcher, and the catcher is the most recently established catcher with such a tag, then no futher forms are evaluated, and the results specified by the thrower are returned by **catch**.

[*Special form*]

unwind-protect *protected-form {cleanup-form}** => *protected-form-results*

This special form ensures that evaluation of the *protected-form* will be followed by the evaluation of each *cleanup-form* (in order from left to right), whether *protected-form* returns normally or is exited via a **return**, a **throw**, a **go**, or an error.

unwind-protect returns the values of the *protected-form*.

Implementation note: Events which cause a stack-group reset (e.g., a stack-overflow error, a cons-space-full error) cause an exit from the protected form without evaluating the cleanup forms.

Examples:

```
(let ((foo '(1)))
    (catch 'tag (unwind-protect
                    (progn
                        (setf foo
                              (cons 2 foo))
                        (throw 'tag nil))
                    (setf foo (cons 3 foo))))
    foo) => (3 2 1)
```

[*Special form*]

throw *Tag result*

This special form (sometimes called a *thrower*) performs a non-local transfer of control (a *throw*) to the most recently established *catcher* whose tag is **eq** to the tag produced by evaluating *tag*.

A catcher with a given tag is established by the evaluation of the tag form of a **catch** form.

The *result* form is evaluated before the transfer of control takes place. The values of the *result* form are returned by the **catch** form that established the catcher which caught the throw.

Within the dynamic extent of the catcher, any dynamic bindings are undone and any **unwind-protect** cleanup-forms are evaluated.

7.11 Closures

In GCLISP, functions can refer to variables that were not bound by (or in) the function. These variables are called *free* or *global* variables. For example, the function

```
(defun foo (a)
   (+ a *b*))
```

has one free variable: ***b***. (Note: By convention, free variables begin and end with an asterisk.) Normally, the value of a free variable that is referenced within a function depends on the binding environment that exists at the time the function is *called*. Thus, the value of ***b*** depends on the binding environment that exists when **foo** is called.

In COMMON LISP, free variables are handled differently. Normally, the value of a free variable that is referenced within a function depends on the binding environment that existed at the time the function was *defined*. This kind of function is called a *closure* since the free variable bindings are *enclosed* with the function.

In GCLISP, the function **closure** allows the user to create a closure. The closure can enclose with a function some or all of the bindings of the free variables occurring in that function. A closure can be used just like any other kind of functional object (i.e., it can be **funcall**'ed and **apply**'ed.)

[*Function*]

closure *variable-list function* => *closure*

This function creates and returns a *closure* that encloses the current binding of each of the variables in the *variable-list* with *function*.

Each element of the list *variable-list* must be a bound variable. **closure** returns a *closure* object that contains *function* and a *copy* of the bindings of the variables in *variable-list*. (This implies that two closures cannot *share* variables.)

When a closure is applied, the enclosed bindings are temporarily established and then the function which was *closed over* is applied. After the application, the enclosed bindings are updated with their current values. (Note: This implies that a recursive invocation of a closure will re-establish the enclosed bindings and hence, the enclosed bindings' values at the time of the call will not be visible.)

Examples:

```
(let ((*b* 1))
    (labels ((func (a) (+ a *b*)))
        (setf clo (let ((*b* 10))
                        (closure '(*b*) #'func))))
    (funcall clo 2)) => 12
```

7.12 Stack Groups

A *stack group* is a functional object which contains its own evaluation state and binding-environment. It has most of the characteristics of a *task* or a *process*. GCLISP stack groups were inspired by and are very similar to ZETALISP stack groups.

When GCLISP is initialized, the Top-Level read-eval-print loop is associated with a stack group. In order to perform an evaluation independent of this initial stack group, one must create a new stack group (using **make-stack-group**), give it an an initial function call to evaluate (using **stack-group-preset**), and then start the new stack group (using, for example, **stack-group-resume**).

Starting or continuing the computation of a stack group is called *resuming*. Resuming a stack group *suspends* the current stack group (hereafter called *c*) and continues the computation of the suspended stack group (hereafter called *s*) at the point *s* was last suspended (or starts the computation if the stack group was just preset). Resuming is also called "switching stack groups".

7.12.1 Stack Group Structure

A stack group contains two stacks (hence the name *stack group*). One stack represents the stack group's execution state (i.e., the state of the computation). This stack has historically been called the *regular PDL* (PDL stands for *Push Down List*, and is pronounced like "piddle"). The other stack represents the stack group's binding-environment. This stack has historically been called the *special PDL*. (Speaking precisely, the special PDL contains all *shadowed* (i.e., saved) variable bindings.) When creating a stack group, using **make-stack-group**, one may specify the size of either PDL.

In addition to the two stacks, a stack group has a *state* (which determines its resumability), a *name* (used in the stack group's printed representation), a *resumer* (another stack group which this stack group can resume), and other internal state information. The state, name, and resumer of a stack group are discussed below.

At any given time, a stack group is in one of the following *states*:

active — The stack group is executing. Only one stack group may be in this state at any given moment; this stack group is called the *current* stack group.

resumable — The stack group is not currently executing; the evaluation represented by the stack group is suspended. The evaluation will continue when the stack group is resumed.

exhausted — The stack group is not currently executing; the evaluation represented by the stack group is finished (i.e., the stack group's initial function call has been completely evaluated). The stack group cannot be resumed (but it may be reset by **stack-group-preset**).

broken — The stack group is not currently executing; the evaluation represented by the stack group signalled an error which is waiting to be handled. The stack group cannot be resumed (but it may be reset by **stack-group-preset**).

The user cannot directly set the state of a stack group; however, it can be displayed using the function **describe**.

The *name* of a stack group is used in the printed representation of the stack group. It is supplied by the user when the stack group is created. It cannot be changed by the user.

If *s* is resumed when *c* calls the function **stack-group-return**, then *s* is said to be the *resumer* of *c*. Each stack group has a cell which contains its resumer. If a stack group has no resumer, this cell contains nil. The user cannot directly modify or display the resumer of a stack group, but it is set when a stack group is invoked as a function.

7.12.2 Creating and Initializing a Stack Group

[*Function*]

make-stack-group *name* **&key :regular-pdl-size :special-pdl-size** => *stack-group*

This function creates and returns a stack group named *name*.

name may be either a symbol or a string; it is used in the printed representation of the new stack group. **:regular-pdl-size** and **:special-pdl-size** must be non-negative integers; they specify the regular and special stack sizes, respectively. They default to 200 and 500 double-words (32 bits), respectively.

The internal state information of the created stack group is undefined; it must be initialized by the function **stack-group-preset.**

[*Function*]

stack-group-preset *stack-group function* **&rest** *arguments* => *stack-group*

This function initializes *stack-group* so that when it is resumed, *function* is applied to *arguments*.

function must be an object which is acceptable to apply. **stack-group-preset** clears both stacks (i.e., the regular and special PDLs), sets the state to *resumable*, and *conceptually* makes

(*function* . *arguments*)

the initial function call.

stack-group may be in any state except *active*; but, if *stack-group* is not in the *exhausted* state, its current evaluation is abandoned without any clean-up (i.e., **unwind-protect** is not honored, but bindings are undone).

The initialized stack group is returned.

[*Function*]

stack-group-unwind

This function resets the currently active stack group.

First, both stacks (the regular and special PDL's) are cleared. As they are being cleared, all bindings are undone and all **unwind-protect** *cleanup* forms are evaluated (in the correct binding environment).

Secondly, the function call **(top-level)** is made the initial function call of the stack group.

Finally, the stack group is resumed.

Note that **stack-group-unwind** never returns.

7.12.3 Resuming a Stack Group

At any given time, only one stack group is *active*, (i.e., in the midst of an evaluation). This active stack group is called the *current* stack group. All other stack groups are either suspended (i.e., in the *resumable* state), exhausted, or broken.

The current stack group (*c*) can resume a suspended stack group (*s*) in one of three ways:

- *c* can call (or apply) *s* as a function of one argument.
- *c* can call the function **stack-group-return.** In this case, *s* must be the stack group that invoked *c* as a function (i.e., *c*'s resumer).
- *c* can call the function **stack-group-resume** with *s* as the first argument.

In addition to resuming *s*, these three methods allow *c* to transmit an object to *s* (if *s* was just preset, the transmitted object is ignored). Each of these three function calls does not return until its associated stack-group is resumed. Each function returns the object received by its stack group.

An example of resuming a stack group by invoking it as a function is the following:

(funcall *sg trans-obj***)**

When this function call is evaluated, the following occurs:

1. The state of *sg* is tested. If it is not *resumable*, an error is signalled. (Note that this implies that the current stack group cannot resume itself.)

2. The current stack group (*c*) is suspended (i.e., put into the *resumable* state).

3. *c* is made the resumer of *sg*. Thus, for example, if *sg* evaluates

   (stack-group-return *recv-obj***)**

   c will be resumed and the function call that resumed *sg* will return *recv-obj*. (Note that neither the function **stack-group-resume** nor the function **stack-group-return** affect the resumer cell of their own stack group or the resumer cell of the stack group they resume.)

4. The *sg* stack group is resumed (i.e., put into the *active* state). If *sg* had been suspended by its evaluation of one of the above function calls (e.g., **stack-group-resume**), then that function call will return *trans-obj* as its value.

[*Function*]

stack-group-resume *stack-group object* => *received-object*

This function resumes *stack-group*, transmitting *object* in the process.

stack-group must be in the *resumable* state.

stack-group-resume returns when its stack group (i.e., the stack group that was active when **stack-group-resume** was called) is resumed. It returns whatever object is received by its stack group.

[*Function*]

stack-group-return *object* => *received-object*

This function resumes the current stack group's *resumer*, tranmitting *object* to it.

The current stack group must have a resumer (e.g., the current stack group was resumed by being called as a function).

stack-group-return returns when its stack group (i.e., the stack group that was active when **stack-group-return** was called) is resumed. It returns whatever object is received by its stack group.

Besides the normal ways of switching stack groups discussed above, various *events* can cause a stack group switch.

One type of stack switching event is an asynchronous event (i.e., an interrupt), such as a clock, garbage-collection, or break event. For example, when a clock event occurs, the value of the variable ***clock-event*** is funcall'ed with some argument. If the value of such a variable is a stack group, a stack group switch will occur.

Note also that certain severe errors (e.g., stack overflow, cons space full, etc.) cause the current stack group to be reset. Conceptually, resetting the current stack group (*c*) involves the following steps:

1. **(stack-group-preset** *c* **'top-level)**

2. *c* is resumed.

7.12.4 Dynamic Bindings and Stack Groups

Since each stack group contains its own binding environment, the dynamic bindings of the current stack group (*c*) are not available to the stack group which *c* resumes. This includes the bindings of such important variables as ***terminal-io***. (Note that although bindings are not shared between stack groups, the global value of a variable is visible to all stack groups.)

Thus, the initial function of a stack group should provide a means to pass the current value of each important dynamic variable. One way to do this is to pass such values as arguments to the initial function. Another way is to make the initial function a closure that closes over the important variables.

7.12.5 Stack Group Variables

[*Variable*]

initial-stack-group => *stack-group*

The value of this variable is the stack group which is created when GCLISP is initialized.

[*Variable*]

current-stack-group => *stack-group*

The value of this variable is the stack group which is currently active. It is automatically updated when a stack group switch is performed. The user should not alter the value of this variable.

Chapter 8

Macros

A *macro call* is a LISP form which is transformed into another (usually more complicated) LISP form before being evaluated. A *macro definition* (often just called a *macro*) defines a mapping between macro-call forms and their *expansions*. The actual transformation is performed by an *expansion function* that is defined by the macro definition.

The GCLISP implementation of macros adheres quite closely to the COMMON LISP specification. The major differences are as follows:

1. The **&key, allow-other-keys,** and **environment** lambda-list keywords are not currently supported.

2. An embedded lambda-list must not contain any lambda-list keywords.

3. The expansion functions **macroexpand** and **macroexpand-1** do not take a lexical environment as a second argument in GCLISP. This makes sense because GCLISP does not currently support lexical scoping.

4. Since there is considerable overhead involved in macro expansion (as compared to a simple function call), GCLISP replaces (using **rplacb**) a macro call with its expansion as part of the expansion process. In LISP jargon, this type of macro is called a *displacing* macro.

An in-depth explanation of macros may be found in the CLRM.

8.1 Macro Definition

[*Function*]

macro-function *symbol* => *nil/expansion-function*

This function is a predicate which is true if and only if the function definition of *symbol* is a macro definition.

The argument must be a symbol. Instead of returning t to indicate true, the macro expansion function is returned.

[*Macro*]

defmacro *Name Macro-lambda-list*
*{Declaration | Doc-string}**
*{Form}** => *name*

This macro makes the function definition of *name* a macro definition. It also creates the associated expansion function.

name must be a symbol that does not name a special form.

macro-lambda-list is an *extended* lambda-list, similar to the lambda-list of a lambda-expression. It is matched against the rest of the macro-call form.

The *macro-lambda-list* may contain the lambda-list keywords: **&optional**, **&rest**, and **&aux**. Two additional lambda-list keywords are allowed: **&body** and **&whole**. **&body** is synonymous with **&rest** (but more meaningful in some cases). **&whole** binds the variable which follows it to the entire macro-call form. If **&whole** is used, it must be the first element of a lambda-list.

Also, any place where a normal lambda-list allows a parameter name, *macro-lambda-list* allows an extended lambda-list. Each embedded lambda-list is matched against a corresponding sub-form of the macro-call form.

Finally, the *macro-lambda-list* may be a dotted-list with a parameter name to the right of the dot. This is identical to ending the list with **&rest** followed by the parameter name.

The forms constitute the body of the expansion function.

Compatibility note: The **&key**, **allow-other-keys**, and **environment** lambda-list keywords are not currently supported. An embedded lambda-list must not contain any lambda-list keywords.

[*Special form*]

macro *Name* (*Var*) {*Form*}* => *name*

This special form is the macro definition primitive.

8.2 Macro Expansion

[*Function*]

macroexpand *form* => *macro-expansion boolean*

This function repeatedly expands *form* until the resulting form is no longer a macro-call form. It then returns the resulting form and either **t** or **nil** depending upon whether or not the original *form* was a macro call form or not, respectively.

In effect, **macroexpand** repeatedly calls **macroexpand-1** until the resulting form is no longer a macro call form.

[*Function*]

macroexpand-1 *form* => *macro-expansion boolean*

This function attempts to expand a macro call form into its macro expansion.

form is checked for being a macro call form. (A macro call form is a list whose **first** element is a symbol that has a macro definition associated with it.)

If *form* is a macro call form, the expansion function associated with the macro is called with *form* as its only argument. The result (the expansion of the macro call) and the symbol **t** are returned by **macroexpand-1**.

Otherwise, *form* is not a macro call and *form* and **nil** are returned.

Chapter 9

Declarations

A declaration associates an entity with information which may be helpful or necessary in processing that entity.

Traditionally, in LISP, the information needed to interpret a given entity is manifested by the entity itself or is manifest within its context. Thus declarations are purely optional, and typically are only used to provide information to a compiler so that a program can be compiled more efficiently.

In COMMON LISP, there is one type of declaration that is not optional: **special** declarations. This is due to the fact that COMMON LISP specifies that local variables be lexically scoped in both interpreted and compiled programs.

In GCLISP, since lexical scoping is not currently supported and no compiler is available, all declarations are completely optional, and in fact, are totally ignored by the interpreter.

Programmers who wish to transport programs written in GCLISP to other COMMON LISP implementations should adhere to the COMMON LISP rules on declarations (e.g., all global variables should be declared *special*.

9.1 Declaration Syntax

[*Special form*]

declare {*Decl-spec*}* => *nil*

This special form has no effect; it does not examine or evaluate any of its arguments, and it returns nil.

It is included for compatibility with other COMMON LISP implementations.

Compatibility note: The **special** declaration specifier has no effect on the interpreter. Also, declarations (i.e., **declare** special forms) *are* evaluated by the interpreter, but they have no effect.

9.2 Declaration Specifiers

Currently, all declaration specifiers (including special) are ignored by the GCLISP interpreter.

9.3 Type Declaration for Forms

GCLISP does not currently support type declarations for forms.

Chapter 10

Symbols

Next to lists, symbols are the most fundamental LISP objects. They are used to represent entities with various properties, to name variables, and to name functions.

Each GCLISP symbol has the following components (sometimes called *cells*):

value
: The current value of the variable named by the symbol. The current value may be undefined.

property list
: A list which contains property/value pairs.

function definition
: The current function (or macro, or special form) definition of the symbol. The current function definition may be undefined.

package
: The package that *owns* this symbol (i.e., the symbol's *home* package).

In addition, each symbol has a *print name*. The print name is a sequence of characters which uniquely identify a symbol within a package.

Compatibility note: A symbol's print name is not stored as an object of type **string**. Thus, functions which return a symbol's print name as a string (e.g., **symbol-name**) always create a string.

10.1 The Property List

[*Function*]

get *symbol indicator* **&optional** *default* => *property-value*

This function returns the *property-value* associated with *indicator* on the property list of *symbol*.

If there is no such *indicator* on *symbol*'s property list (i.e. *indicator* is not **eq** to some indicator on the property list), **default** (which defaults to **nil**) is returned.

Note that there is no way to distinguish between a property whose value is **default**, and a non-existent property.

[*Function*]

remprop *symbol indicator* **=>** *boolean*

This function removes the property, whose indicator is **eq** to *indicator*, from the property list of *symbol*.

If a property on the property list of *symbol* does have an indicator **eq** to *indicator*, the property's indicator and value are *spliced* out of the property list, and **remprop** returns **t**; otherwise, the property list is unaffected, and **nil** is returned.

(remprop *x y***) <=> (remf (symbol-plist** *x***)** *y***)**

[*Function*]

symbol-plist *Symbol* **=>** *property-list*

This function returns the property list of *symbol*. Note that the actual property list (not a copy) is returned.

Examples:

```
(progn (setf (get 'foo 'frob) 7)
       (symbol-plist 'foo)) => (frob 7)
```

[*Macro*]

getf *place indicator* **&optional** *default* => *property-value*

This macro returns the *property-value* associated with *indicator* on the property list returned by *place*.

getf is the generalized variable version of **get**.

If there is no such *indicator* on the property list (i.e. *indicator* is not **eq** to some indicator on the property list), **default** (which defaults to **nil**) is returned.

place must evaluate to a list.

Note that there is no way to distinguish between a property whose value is **default**, and a non-existent property.

Compatibility note: getf is not acceptable as a place to **setf**.

[*Function*]

remf *place indicator* => *boolean*

This function removes the property, whose indicator is **eq** to *indicator*, from the property list named by *place*.

remf is the generalized variable version of **remprop**.

place must be a place acceptable to **setf**.

If a property on the property list named by *place* has an indicator **eq** to *indicator*, then the property's indicator and value are *spliced* out of the property list, and **remf** returns **t**; otherwise, the property list is unaffected, and **nil** is returned.

[*Function*]

get-properties *place indicator-list*
    => *indicator value nil/property-list-tail*

This function searches the property-list stored at *place* for an indicator that is **eq** to a member of *indicator-list*.

10.2 The Print Name

[*Function*]

symbol-name *symbol* => *print-name*

This function creates and returns a string that contains the print name of *symbol*.

[*Function*]

samepnamep *symbol1 symbol2* => *boolean*

This function is a predicate which is true if and only if the print name of *symbol1* is **equal** to the print name of *symbol2*.

10.3 Creating Symbols

[*Function*]

make-symbol *print-name* => *new-symbol*

Creates and returns an uninterned symbol whose print name is *print-name*.

print-name must be a string.

The new symbol has no value, no functional definition, an empty property list, and no home package.

[*Function*]

copy-symbol *symbol* **&optional** *copy-props-p* **=>** *new-symbol*

This function creates and returns an uninterned symbol with the same print name as *symbol*.

If *copy-props-p* is **nil** (the default), the new symbol will be unbound, have no functional definition, and have an empty property list.

Otherwise, if *copy-props-p* is **t**, the value and function definition of the new symbol will be the same as those of *symbol*, and the property list of new symbol will be a copy of the property list of *symbol*.

[*Function*]

gensym **&optional** *reset* **=>** *new-symbol*

Creates and returns an uninterned symbol with an invented print name.

The invented print name consists of a prefix (originally "G") followed by the decimal representation (without a decimal point) of an integer (originally '0'). Each time (after) **gensym** invents a print name, the integer is incremented by one.

The optional argument, *reset*, if provided, must be a string or a non-negative integer. If *reset* is a string, then the prefix used by **gensym** is changed to that string. If *reset* is a non-negative integer, then the counter used by **gensym** is reset to *reset*.

After resetting the prefix or the counter, **gensym** returns a new symbol (with the new prefix or integer) as it normally does.

Examples:

```
(values (gensym)
        (gensym 64)
        (gensym)
        (gensym "foo-")
        (gensym))
    => g0 g64 g65 foo-66 foo-67
```

[*Function*]

symbol-package *symbol* => *nil/package*

This function returns the home package of *symbol* or **nil** if *symbol* has no home package.

[*Function*]

keywordp *object* => *boolean*

This function is a predicate which is true if and only if *object* is a *keyword* symbol.

Chapter 11

Packages

A package represents a name space (i.e., a mapping from print names to symbols). GCLISP provides all of the essential COMMON LISP package functions and variables (although some less useful ones are not currently implemented).

For an in-depth discussion of the package system, consult the CLRM.

11.1 Consistency Rules

GCLISP adheres to the COMMON LISP consistency rules.

11.2 Package Names

GCLISP package naming adheres to the COMMON LISP specification.

11.3 Translating Strings to Symbols

GCLISP's translation of strings to symbols adheres to the COMMON LISP specification.

11.4 Exporting and Importing Symbols

GCLISP's exporting and importing of symbols adheres to the COMMON LISP specification.

11.5 Name Conflicts

GCLISP's signalling and handling of name conflicts adheres to the COMMON LISP specification.

11.6 Built-in Packages

GCLISP provides all of the COMMON LISP specified packages.

11.7 Package System Functions and Variables

[*Variable*]

package => *current-package*

The value of this variable is the *current* package, i.e., the package which is used by the LISP reader to map a print name to a symbol.

Only objects of type **package** may be assigned to this variable.

Its initial value is the **user** package.

[*Function*]

make-package *package-name* **&key :nicknames :use** => *package*

This function creates and returns a new package named *package-name*.

package-name must be an acceptable package name (i.e., a string or a symbol which is not used as a name for an existing package).

:nicknames must be a list of acceptable package names. Each nickname may be used as an alternative name for the package. **:nicknames** defaults to the empty list.

:use must be a list of packages (or their names). The external symbols of each of the packages is inherited by the new package. **:use** defaults to a list of one package, the LISP package.

[*Function*]

in-package *package-name* **&key :nicknames :use** => *package*

If a package named *package-name* already exists, **in-package** returns that package, adding any new names in the **:nicknames** list or new packages in the **:use** list.

Otherwise, if no such package exists, **in-package** creates and returns a new package just like **make-package**.

In either case, **in-package** also assigns the package that it returns to the variable ***package***.

in-package is intended to be used at the start of a file containing a subsystem that is to be placed into its own package.

[*Function*]

find-package *name* => *nil/package*

This function is a predicate which is true if and only if *name* is the name or nickname of some package.

name must be either a string or a symbol (whose print-name is used). Names are compared as if by **string=**.

Instead of returning **t** to indicate true, the package named by *name* is returned.

[*Function*]

package-name *package* => *package-name*

This function returns the name (a string) of *package*.

The argument must be a package (not a package name).

[*Function*]

package-nicknames *package* => *nickname-list*

This function returns a list of nicknames (strings) of *package*.

The argument must be a package (not a package name).

[*Function*]

package-use-list *package* => *used-packages-list*

This function returns a list of packages used by *package*.

The argument may be a package or the name of a package.

[*Function*]

package-used-by-list *package* => *users-of-package-list*

This function returns a list of packages that use *package*.

The argument may be a package or the name of a package.

[*Function*]

package-shadowing-symbols *package*
=> *shadowing-symbols-list*

This function returns *package*'s list of shadowing symbols.

The argument may be a package or the name of a package.

Shadowing symbols are declared by the functions **shadow** and **shadowing-import**.

[*Function*]

list-all-packages => *all-packages-list*

This function returns a list of all existing packages.

[*Function*]

intern *string* **&optional** *package* => *symbol existed-p*

This function returns both a symbol whose print name is *string* and which is present in *package*, and a value indicating whether or not the symbol was created by this invocation of **intern**.

The first argument must be a string (it may not be a symbol).

The second argument may be a package or the name of a package (string or symbol). It defaults to the current package.

package is first searched for a symbol whose print name is **string=** to *string*. (The search includes inherited symbols.) If one is not found, a symbol is created (as if by **make-symbol**), with *string* as its print name, and is made present in *package*. This newly created symbol is then returned with the symbol **nil**.

Otherwise, if such a symbol is accessible in *package*, it is returned with one of the following keywords:

:internal The symbol was present in *package* as an internal symbol.

:external The symbol was present in *package* as an external symbol.

:inherited The symbol was inherited by *package* (and was therefore accessable as an internal symbol in *package*).

[*Function*]

find-symbol *string* **&optional** *package*
    => *nil/symbol existed-p*

This function tests whether a symbol (call it *s*) whose print name is *string* is accessible in *package*.

find-symbol is identical to **intern** except that **find-symbol** never creates a new symbol. Instead, if *s* is not accessible in *package*, both values returned by **find-package** are **nil**. Otherwise, *s* is returned as the first value and the second value is as specified for **intern**.

[*Function*]

unintern *symbol* **&optional** *package* => *boolean*

This function removes *symbol* from *package*.

The first argument must be a symbol.

The second argument may be a package or the name of a package (string or symbol). It defaults to the current package.

If *symbol* is present in *package*, it is removed from *package* and also from *package*'s shadowing-symbols list (if it appears there). (Note that uninterning a shadowing symbol can uncover a name conflict.) In addition, if *package* was the home package of *symbol*, *symbol* is made *homeless*. **unintern** then returns **t**.

Otherwise, if *symbol* is not present in *package*, **unintern** returns **nil**.

[*Function*]

export *symbols* **&optional** *package* => *t*

This function causes each of the *symbols* present in *package* to be accessible as external symbols.

symbols may be a single symbol or a list of symbols (with **nil** representing the empty list). The symbols should be accessible in *package*, an error is signalled if they are not.

All packages which use *package* are checked for name conflicts.

package may be a package or the name of a package (string or symbol). It defaults to the current package.

[*Function*]

unexport *symbols* **&optional** *package* => *t*

This function changes each of the *symbols* which are present in *package* as external symbols to internal symbols.

symbols may be a single symbol or a list of symbols (with **nil** representing the empty list). The symbols should be accessible in *package*, an error is signalled if they are not. Symbols that are accessible in *package* but not external are left unchanged. It is an error to unexport a symbol from the keyword package.

package may be a package or the name of a package (string or symbol). It defaults to the current package.

[*Function*]

import *symbols* **&optional** *package* => *t*

This function causes each of the *symbols* to be present in *package* as an internal symbol.

symbols may be a single symbol or a list of symbols (with **nil** representing the empty list). **import** checks each symbol (call it *s*) for a name conflict as follows: If *s* is already present in *package*, **import** does not affect it; if a distinct symbol with the same print name as *s* is accessible in *package*, **import** signals an error (even if the distinct symbol is a shadowing symbol).

package may be a package or the name of a package (string or symbol). It defaults to the current package.

[*Function*]

shadowing-import *symbols* **&optional** *package* => *t*

This function causes each of the *symbols* to be present in *package* as an internal symbol.

It is identical to **import**, except that each symbol is put on *package*'s shadowing-symbols list and no name conflict error is ever signalled.

If a symbol (call it *s*) that is present in *package* has the same print name as, but is distinct from, a symbol being imported by **shadowing-import**, then *s* is uninterned.

package may be a package or the name of a package (string or symbol). It defaults to the current package.

[*Function*]

shadow *symbols* **&optional** *package* => *t*

This function causes symbols with the same print names as *symbols* to be present in *package* as internal symbols and to be placed on the shadowing-symbols list of *package*. A name conflict error is never signalled.

symbols may be a single symbol or a list of symbols (with **nil** representing the empty list). **shadow** gets the print name of each symbol and checks to see if a symbol (call it *s*) with that print name is present in or inherited by *package*.

If *s* is present in *package*, **shadow** simply adds *s* to the shadowing-symbols list of *package*.

Otherwise, if *s* is inherited by *package*, a new symbol with the same print name as *s* is created and made present in *package* as an internal symbol. The new symbol is also added to the shadowing-symbols list of *package*.

package may be a package or the name of a package (string or symbol). It defaults to the current package.

[*Function*]

use-package *packages-to-use* **&optional** *package* => *t*

This function adds each of the *packages-to-use* to the use-list of *package*.

packages-to-use must be a package or package name, or a list of such. *package* inherits each package's external symbols, i.e. the external symbols will be accessible in *package* as internal symbols. Each external symbol is checked for name conflicts.

package may be a package or the name of a package (string or symbol). It defaults to the current package.

[*Function*]

unuse-package *packages-to-unuse* **&optional** *package* => *t*

This function removes each of the *packages-to-unuse* from the use-list of *package*.

packages-to-use must be a package or package name, or a list of such. *package* no longer inherits each package's external symbols, i.e., the external symbols are not accessible in *package* as internal symbols.

package may be a package or the name of a package (string or symbol). It defaults to the current package.

[*Macro*]

do-symbols (*Var* [*package-form* [*Result-form*]])
    {*declaration*}* {*Tag* | *Statement*}*
    => *result-form-results*

This macro provides simple iteration over the symbols that are accessible in a package.

var must be a symbol. *package-form* must evaluate to a package. *result-form* must be a valid form. The rest of the macro call is treated as an implicit tagbody.

The tagbody and the *result-form* are within an implicit **block** and within an environment in which *var* is bound. For each symbol that is accessible in the package, *var* is assigned that variable and the tagbody is evaluated. (**return** may be used to exit the iteration at any time.)

After the iteration is complete, the *result-form* is evaluated (with *var* bound to **nil**) and its results are returned by

do-symbols.

[*Macro*]

do-external-symbols (*Var* [*package-form* [*result-form*]]) {*declaration*}* {*Tag* | *statement*}*
=> *result-form-results*

This macro provides simple iteration over the symbols that are present in a package as external symbols. In all other respects it is identical to **do-symbols.**

[*Macro*]

do-all-symbols (*Var* [*result-form*]) {*declaration*}* {*Tag* | *statement*}*
=> *result-form-results*

This macro provides simple iteration over all the symbols in every package. It functions similarly to **do-symbols.** Note that symbols which are present in more than one package will be processed more than once.

11.8 Modules

Modules are not currently supported.

Chapter 12

Numbers

GCLISP provides the following distinct types of numbers: fixnums, single-floats, and double-floats.

Fixnums are the only type of integer currently supported by GCLISP. A GCLISP fixnum is represented in two's complement notation and may range from -2^{15} to $2^{15}-1$ (thus a fixnum occupies 16 bits). There are no *pointers* to fixnums, they are directly represented by a variant type of pointer. An error is signalled if an integer computation produces a result outside this range.

Two types of floating-point numbers, of different precisions, are provided by GCLisp. Both conform to the IEEE "Proposed Standard for Binary Floating Point Arithmetic." To be precise, a *single* precision floating-point number is represented in Intel 8087 *short real* format, while a *double* precision floating-point number is represented in Intel 8087 *long real* format. Thus, a single precision float occupies 32 bits, can represent 6 to 7 significant digits, and has a range from $8.43*10^{-37}$ to $3.37*10^{38}$, while a double precision float occupies 64 bits, can represent 15 significant digits, and has a range from $4.19*10^{-307}$ to $1.67*10^{308}$. An error is signalled if a floating-point computation causes the exponent to overflow or underflow.

12.1 Precision, Contagion, and Coercion

GCLISP conforms to the COMMON LISP rules of coercion and contagion.

12.2 Predicates on Numbers

Each of the following predicates requires that its argument at least be of type **number**.

[*Function*]

zerop *number* => *boolean*

This function is a predicate which is true if and only if *number* is zero (either of type **integer** or **float**).

The argument must be of type **number**.

[*Function*]

plusp *number* => *boolean*

This function is a predicate which is true if and only if *number* is strictly greater than zero.

The argument must be of type **number**.

[*Function*]

minusp *number* => *boolean*

This function is a predicate which is true if and only if *number* is strictly less than zero.

The argument must be of type **number**.

[*Function*]

oddp *integer* => *boolean*

This function is a predicate which is true if and only if *integer* is odd (not evenly divisible by two).

The argument must be of type **integer**.

[*Function*]

evenp *integer* => *boolean*

This function is a predicate which is true if and only if *integer* is even (evenly divisible by two).

The argument must be of type **integer**.

12.3 Comparisons on Numbers

Each of the following functions requires that its arguments all be of type **number**. The arguments may be of different subtypes; conversions will be performed according to the rules of coercion and contagion.

[*Function*]

= *number* **&rest** *more-numbers* => *boolean*

This function is a predicate which is true if and only if the arguments are all the same number.

Each argument must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

[*Function*]

/= *number* **&rest** *more-numbers* => *boolean*

This function is a predicate which is true if and only if the arguments are all different numbers.

Each argument must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

[*Function*]

< *number* **&rest** *more-numbers* => *boolean*

This function is a predicate which is true if and only if the arguments are numbers which are monotonically increasing from left to right.

Each argument must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

[*Function*]

> *number* **&rest** *more-numbers* => *boolean*

This function is a predicate which is true if and only if the arguments are numbers which are monotonically decreasing from left to right.

Each argument must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

[*Function*]

<= *number* **&rest** *more-numbers* => *boolean*

This function is a predicate which is true if and only if the arguments are numbers which are monotonically nondecreasing from left to right.

Each argument must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

[*Function*]

>= *number* **&rest** *more-numbers* => *boolean*

This function is a predicate which is true if and only if the arguments are numbers which are monotonically nonincreasing from left to right.

Each argument must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

[*Function*]

max *number* **&rest** *more-numbers* => *greatest-number*

This function returns the argument which is greatest (i.e., closest to positive infinity).

All arguments must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

Implementation note: If any of the arguments to **max** is of type **float**, then the result will be of the same type.

[*Function*]

min *number* **&rest** *more-numbers* => *least-number*

This function returns the argument which is least (i.e., closest to negative infinity).

All arguments must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

Implementation note: If any of the arguments to **min** is of type **float**, then the result will be of the same type.

12.3.1 Comparisons on Unsigned Fixnums

Each of the following functions requires two fixnums as arguments. The fixnums are treated as 16-bit unsigned integers.

[*Function*]

<& *unsigned-fixnum-1 unsigned-fixnum-2* => *unsigned-fixnum*

This function is a predicate which is true if and only if *unsigned-fixnum-1* is less than *unsigned-fixnum-2*.

[*Function*]

>& *unsigned-fixnum-1 unsigned-fixnum-2* => *unsigned-fixnum*

This function is a predicate which is true if and only if *unsigned-fixnum-1* is greater than *unsigned-fixnum-2*.

12.4 Arithmetic Operations

Each of the following functions requires that its arguments all be of type **number**. The arguments may be of different subtypes; conversions will be performed according to the rules of coercion and contagion.

[*Function*]

+ **&rest** *numbers* => *sum*

This function returns the arithmetic **sum** of its arguments. If no arguments are given, the integer 0 (the identity for this operation) is returned.

All of the arguments must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

[*Function*]

- *number* **&rest** *more-numbers* => *difference*

When given one argument, this function returns the negative of that argument.

When given two or more arguments, this function successively subtracts the second through the last argument from the first argument and returns the result.

All of the arguments must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

```
(- n1 n2 n3 ... nn-1 nn)
    <=> (- (- ... (- (- n1 n2)
                      n3) ... nn-1) nn)
```

[*Function*]

***** **&rest** *numbers* => *product*

This function returns the arithmetic product of its arguments. If given no arguments, the integer 1 (the identity for this operation) is returned.

All of the arguments must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

[*Function*]

/ *number* **&rest** *more-numbers* => *quotient*

When given one argument, this function returns the reciprocal of that argument.

When given two or more arguments, this function successively divides the second through the last argument into the first argument and returns the result.

All of the arguments must be of type **number**. Arguments of different subtypes are converted according to the rules of coercion and contagion.

```
(/ n1 n2 n3 ... nn-1 nn)
    <=> (/ (/ ... (/ (/ n1 n2)
                      n3) ... nn-1) nn)
```

[*Function*]

1+ *number* => *successor*

This function returns the sum of *number* and one. It is equivalent to,

(+ *number* **1).**

[*Function*]

1- *number* => *predecessor*

This function returns the difference of *number* and one. It is equivalent to,

(- *number* **1).**

[*Macro*]

incf *place* [*delta*] => *incremented-result*

This macro adds the value of *delta* to the number stored at *place*, stores this sum back into *place*, and returns the sum.

place must be a form acceptable as a generalized variable to **setf.** The value of the generalized variable named by PLACE must be a number.

If the *delta* argument is given, it must evaluate to a number. If it is not given, *delta* defaults to the integer 1.

[*Macro*]

decf *place* [*delta*] => *decremented-result*

This macro subtracts the value of *delta* from the number stored at *place*, stores this difference back into *place*, and returns the difference.

place must be a form acceptable as a generalized variable to **setf**. The value of the generalized variable named by *place* must be a number.

If the *delta* argument is given, it must evaluate to a number. If it is not given, *delta* defaults to the integer 1.

12.4.1 Unsigned Fixnum Arithmetic

Each of the following functions requires two fixnums as arguments. The fixnums are treated as 16-bit unsigned integers.

[*Function*]

+& *unsigned-fixnum-1 unsigned-fixnum-2* => *unsigned-fixnum*

This function returns the sum of two unsigned fixnums.

[*Function*]

-& *unsigned-fixnum-1 unsigned-fixnum-2* => *unsigned-fixnum*

This function returns the difference of *unsigned-fixnum-1* and *unsigned-fixnum-2*.

[*Function*]

***&** *unsigned-fixnum-1 unsigned-fixnum-2* => *unsigned-fixnum*

This function returns the product of two unsigned fixnums.

[*Function*]

/& *unsigned-fixnum-1 unsigned-fixnum-2* **=>** *unsigned-fixnum*

This function returns the quotient of *unsigned-fixnum-1* and *unsigned-fixnum-2*.

12.5 Irrational and Trancendental Functions

All of the following functions require the presence of the Intel 8087 Numeric Processor Extension.

12.5.1 Exponential and Logarithmic Functions

[*Function*]

exp *number* **=>** *number*

This function returns *e* raised to the power *number*, where *e* is the base of the natural logarithms.

[*Function*]

expt *base-number power-number* **=>** *number*

This function returns *base-number* raised to the power *power-number*. If both arguments are integers the result will be an integer; otherwise, a floating-point number may result.

[*Function*]

log *number* **&optional** *base* **=>** *number*

This function returns the logarithm of *number* in the base *base* (which defaults to 0).

[*Function*]

sqrt *number* => *number*

This function returns the principle square root of *number*.

12.5.2 Trigonometric and Related Functions

[*Function*]

abs *number* => *number*

This function returns the absolute value of *number*.

```
(abs number)
    <=> (if (minusp number)
            (- number)
            number)
```

[*Function*]

signum *number* => *sign-number*

This function will return one of the numbers, -1, 0, or 1, depending on whether *number* is negative, zero, or positive, respectively.

```
(signum number)
    <=> (if (zerop number)
            number
            (/ number (abs number)))
```

[*Function*]

sin *radians* => *number*

This function returns the sine of *radians*.

[*Function*]

cos *radians* => *number*

This function returns the cosine of *radians*.

[*Function*]

tan *radians* => *number*

This function returns the tangent of *radians*.

[*Function*]

atan *y* &optional *x* => *radians*

This function returns an arc tangent in radians. If given one argument, **atan** returns its arc tangent. Given two arguments, the arc tangent of *y/x* is returned.

12.5.3 Branch Cuts, Principle Values, and Boundary Conditions in the Complex Plane

GCLISP does not currently support complex numbers.

12.6 Type Conversions and Component Extractions on Numbers

[*Function*]

float *number* **&optional** *template* => *float-number*

This function converts *number* to a floating point number, i.e., an object of type **float**.

number must be of type **number**. The optional argument, *template*, must be of type **float**. If *template* is not given, then *number* is converted to a floating point number of type **single-float**; unless *number* is already of type **float**, in which case it is simply returned. If *template* is given, then *number* is converted to a floating point number of the same type as *template* (even if *number* was already of type **float**).

[*Function*]

floor *number* => *integer*

This function returns the greatest integer that is less than or equal to *number*, i.e., it truncates toward negative infinity.

[*Function*]

ceiling *number* => *integer*

This function returns the least integer that is not less than *number*, i.e., it truncates toward positive infinity.

[*Function*]

truncate *number* **&optional** *divisor* => *number*

This function converts a specified number to an integer by truncating towards zero.

number must be of type *number*. If the optional argument, *divisor*, is not given, it defaults to the number 1 (of the same type as *number*).

Given two arguments, *n* and *d*, **truncate** returns two values, *q* and *r*, such that,

$q * d + r = n$.

Where q is an integer such that **(ABS** q**)** < **(ABS** n**)**, and r is a number whose type is either **integer** (if both n and d are integers) or **float** (if either n or d is of type **float**).

[*Function*]

round *number* => *integer*

This function returns the integer that is closest to *number*. If two integers are equally close, the even integer is returned.

[*Function*]

mod *integer divisor* => *integer*

This function returns the smallest integer remainder of *integer/divisor* that is of the same sign as *divisor*. Both arguments must be integers.

12.7 Logical Operations on Numbers

The following functions accept only integers (i.e., fixnums) as arguments. They all treat an integer as a sequence of bits which represents the value of the integer in two's-complement notation.

[*Function*]

logior **&rest** *integers* => *integer*

This function returns an integer which is the result of a bit-wise logical *inclusive or* of all of its arguments. If no arguments are given, zero (the identity for this operation) is returned.

[*Function*]

logxor **&rest** *integers* **=>** *integer*

This function returns an integer which is the result of a bit-wise logical *exclusive or* of all of its arguments. If no arguments are given, zero (the identity for this operation) is returned.

[*Function*]

logand **&rest** *integers* **=>** *integer*

This function returns an integer which is the result of a bit-wise logical *and* of all of its arguments. If no arguments are given, -1 (the identity for this operation) is returned.

[*Function*]

logeqv **&rest** *integers* **=>** *integer*

This function returns an integer which is the result of a bit-wise logical *equivalence* (i.e., the *exclusive nor*) of all of its arguments. If no arguments are given, -1 (the identity for this operation) is returned.

[*Function*]

lognot *integer* **=>** *integer*

This function returns an integer which is the bit-wise logical *not* of INTEGER.

```
(logbitp index (lognot integer))
    <=> (not (logbitp index integer))
```

[*Function*]

logtest *integer1 integer2* => *boolean*

This function is a predicate which is true if and only if there is a bit in *integer1* and a bit in the same position in *integer2* which are both one-bits.

```
(logtest integer1 integer2)
    <=> (not
          (zerop
            (logand integer1 integer2)))
```

[*Function*]

logbitp *index integer* => *boolean*

This function is a predicate which is true if and only if the *index*th bit of *integer* is a one-bit.

```
(logbitp index integer)
    <=> (not (zerop (logand integer
                             (ash 1 index))))
```

[*Function*]

ash *integer count* => *integer*

This function arithmetically shifts *integer* by *count* bit positions.

If *count* is a non-negative integer, *integer* is shifted *count* positions to the left (filling with zeros on the right and discarding bits on the left).

If *count* is a negative integer, *integer* is shifted *count* positions to the right (copying the sign bit on the left and discarding bits on the right).

Compatibility note: Since integers are of fixed size, an arithmetic shift left can cause the sign to change.

[*Function*]

lsh *integer count* => *integer*

This function logically shifts *integer* by *count* bit positions.

If *count* is a non-negative integer, *integer* is shifted *count* positions to the left (filling with zeros on the right and discarding bits on the left).

If *count* is a negative integer, *integer* is shifted *count* positions to the right (filling with zeros on the left and discarding bits on the right).

12.8 Byte Manipulation Functions

Byte manipulation functions are not currently supported.

12.9 Random Numbers

Random numbers are not currently supported.

12.10 Implementation Parameters

The float and fixnum parameters are not currently supplied.

Chapter 13

Characters

In GCLISP, the type **character** is a subtype of the type **fixnum**. That is, GCLISP internally represents characters as fixnums in the range 0 (inclusive) through 1024 (exclusive).

13.1 Character Attributes

The current GCLISP **char-code-limit** is 256.

The current GCLISP **char-font-limit** is 1.

The current GCLISP **char-bits-limit** is 4.

13.2 Predicates on Characters

In the following predicates, the argument *char* must be an object of type **character**.

[*Function*]

standard-char-p *char* => *boolean*

This function is a predicate which is true if and only if *char* is a *standard* character.

Any character with non-zero *bits* or *font* attributes is not a standard character.

[*Function*]

alpha-char-p *char* => *boolean*

This function is a predicate which is true if and only if *char* is an alphabetic character.

In the *standard character set*, the letters **A** through **Z** and **a** through **z** are alphabetic.

[*Function*]

upper-case-p *char* => *boolean*

This function is a predicate which is true if and only if *char* is an upper-case (majuscule) character.

In the *standard character set*, the letters **A** through **Z** are upper-case.

[*Function*]

both-case-p *char* => *boolean*

This function is a predicate which is true if and only if either *char* is an upper-case character and it has a corresponding lower-case character; or *char* is a lower-case character and it has a corresponding upper-case character.

In the *standard character set*, the upper-case letters **A** through **Z** have the corresponding lower-case letters **a** through **z**, and vice versa.

[*Function*]

digit-char-p *char* **&optional** *radix* => *weight*

This function is a predicate which is true if and only if *char* is digit of the specified radix.

char must be a character and *radix* must be an integer in the range 2 through 36 (inclusive). If not given, *radix* defaults to 10. If **digit-char-p** is true, it returns the *weight* (an integer) of the digit in the specified *radix*.

In the *standard character set*, the characters **0** through **9** and the alphabetic characters (**A** to **Z**, **a** to **z**) are digits with weights 0 through 9 and 10 through 36 respectively.

[*Function*]

char= *char* **&rest** *more-chars* **=>** *boolean*

This function is a predicate which is true if and only if all of its arguments are all the same character.

[*Function*]

char-equal *char* **&rest** *more-chars* **=>** *boolean*

This function is a predicate which is true if and only if the arguments are all the same character (ignoring differences in case).

[*Function*]

char-lessp *char* **&rest** *more-chars* **=>** *boolean*

This function is a predicate which is true if and only if the arguments are characters which are monotonically increasing from left to right (ignoring differences in case).

13.3 Character Construction and Selection

[*Function*]

char-code *char* **=>** *code*

This function returns the *code* attribute of *char*. *code* will be a non-negative integer less than 256.

[*Function*]

char-bits *char* => *bits*

This function returns the *bits* attribute of *char*. *bits* will be a non-negative integer less than 3.

[*Function*]

code-char *code* **&optional** *bits font* => *character*

This function returns a character object that has the specified *code*, *bits*, and *font* attributes. If such a character object is not valid within the given implementation, **nil** is returned.

The *bits* and *font* attributes default to **0**.

13.4 Character Conversions

[*Function*]

char-upcase *char* => *up-char*

This function attempts to convert *char* to its upper-case equivalent.

If *char* is a lower-case character with an upper case equivalent, that equivalent character is returned; otherwise *char* is returned.

[*Function*]

char-downcase *char* => *low-char*

This function attempts to convert *char* to its lower-case equivalent.

If *char* is an upper-case character with an lower case equivalent, that equivalent character is returned; otherwise *char* is returned.

[*Function*]

char-name *char* => *name*

This function attempts to return the *name* (a string) of *char*.

If there is a *name* for *char*, that *name* is returned; otherwise **nil** is returned.

The standard characters *<newline>* and *<space>* have the respective names **Newline** and **Space**.

[*Function*]

name-char *name* => *character*

This function attempts to return the *character* named by *name* (which may be any object that can be coerced to a string).

If *name* matches the name of some character (using **string-equal**, then that character is returned; otherwise **nil** is returned.

The standard characters *<newline>* and *<space>* have the respective names **Newline** and **Space**.

13.5 Character Control-Bit Functions

[*Function*]

char-bit *char bit-name* => *boolean*

This function returns **t** if the *bit-name* bit is set in *char*, otherwise it returns **nil**.

bit-name must be one of the following: **:control**, or **:meta**.

[*Function*]

set-char-bit *char bit-name new-value* => *char*

This function returns *char* with the *bit-name* bit set (or reset) to *new-value*.

If *new-value* is **nil**, the bit is reset; otherwise, the bit is set.

bit-name must be one of the following: **:control**, or **:meta**.

Chapter 14

Sequences

A *sequence* is an ordered set of elements. Since an object of type **list** or an object of type **vector** (i.e., a one-dimensional array) can be used to represent an ordered set of elements, both types are considered subtypes of the type **sequence**.

There are operations which require only that their argument(s) be an ordered set of elements (i.e., a sequence). Thus, they work equally well on lists or vectors. GCLISP provides some of the most useful generic sequence operations.

14.1 Simple Sequence Functions

[*Function*]

subseq *sequence start* **&optional** *end* **=>** *subsequence*

This function returns a new sequence (of the same type as *sequence*) containing the elements of *sequence* from position *start* (inclusive) to *end* (exclusive).

[*Function*]

length *sequence* **=>** *number-of-elements*

This function returns the number of elements (a non-negative integer) in *sequence*.

If *sequence* is a vector with a fill-pointer, the *active length* of the vector is returned.

[*Function*]

reverse *sequence* => *reverse-sequence*

This function creates and returns a new sequence of the same type as *sequence*, in which the elements of *sequence* are stored in reverse order.

Compatibility note: The *sequence* argument must be a list.

[*Function*]

nreverse *sequence* => *reverse-sequence*

This function returns a sequence in which the elements of *sequence* are stored in reverse order. *sequence* may be altered in the process. The result of **nreverse** may or may not be **eq** to *sequence*.

Compatibility note: The *sequence* argument must be a list.

14.2 Concatenating, Mapping, and Reducing Sequences

The functions in this section currently operate on lists, but not on vectors.

[*Function*]

some *predicate sequence* **&rest** *more-sequences* => *nil/element*

This function maps *predicate* over the sequence arguments. If at some point *predicate* returns a non-nil value, **some** immediately returns that value. If the end of one of the sequences is reached (i.e., *predicate* always returned **nil**), **some** returns **nil**.

Compatibility note: The *sequence* arguments must be lists.

[*Function*]

every *predicate sequence* **&rest** *more-sequences* => *boolean*

This function maps *predicate* over the sequence arguments. If at some point *predicate* returns **nil**, **every** immediately returns **nil**. If the end of one of the sequences is reached (i.e., *predicate* always returned a non-**nil** value), **every** returns **t**.

Compatibility note: The *sequence* arguments must be lists.

14.3 Modifying Sequences

[*Function*]

remove *item sequence* => *new-sequence*

This function returns a copy of *sequence* with all elements **eql** to *item* removed.

remove is the non-destructive counterpart of **delete**.

Compatibility note: The *sequence* argument must be a list.

[*Function*]

remove-if *test sequence* => *new-sequence*

This function returns a copy of *sequence* with all elements that satisfy *test* removed.

remove-if is the non-destructive counterpart of **delete-if**.

Compatibility note: The *sequence* argument must be a list.

[*Function*]

delete *item sequence* **=>** *sequence*

This function returns *sequence* with all elements **eql** to *item* removed.

delete is the destructive counterpart of **remove**.

Compatibility note: The *sequence* argument must be a list.

[*Function*]

delete-if *test sequence* **=>** *sequence*

This function returns *sequence* with all elements that satisfy *test* removed.

delete-if is the destructive counterpart of **remove-if**.

Compatibility note: The *sequence* argument must be a list.

14.4 Searching Sequences for Items

14.5 Sorting and Merging

[*Function*]

sort *sequence predicate* **&key :key** **=>** *sorted-sequence*

This function destructively sorts *sequence* in the order imposed by *predicate* and returns the sorted sequence.

predicate must be a function of two arguments which returns a non-nil value if and only if the first argument is strictly less than the second argument.

:key must be a function of one argument which, when given an element of *sequence*, returns the key for that element. The results of the **:key** function are given to *predicate*.

sort is not guaranteed 'stable'.

Chapter 15

Lists

A *cons* (also called a *dotted-pair*) is a data structure that consists of two components, named after their respective accessor functions: **car** and **cdr** (pronounced like *could-er*). (The two components are also named after the newer and more meaningful accessor functions *first* and *rest*.) The **car** and **cdr** components of a cons are referred to as the "**car** of" and "**cdr** of" the cons, respectively.

A given non-empty list is represented by one or more conses. The **car** of the first cons contains the first element of the list. The **car** of the second cons contains the second element of the list. In general, the **car** of the *n*th cons always contains the *n*th element of a non-empty list. The **cdr** of the first cons contains the second cons (actually, a pointer to it). The **cdr** of the second cons contains the third cons. In general, the **cdr** of the *n*th cons contains the *n*th+1 cons.

Thus, each cons can be viewed as a sublist of the cons that contains it. The **cdr** of the last cons of a list contains an atom (i.e., a non-cons object). If the atom is the symbol **nil**, the list is called a *true* or *ordinary* list; otherwise, the list is called a *dotted* list.

The empty list is represented by the symbol **nil** (which may also be represented by **()**.)

15.1 Conses

[*Function*]

car *list* **=>** *first-element*

This function returns the first element (i.e., the *car*) of *list*.

list must be either a **cons** or **nil** (i.e., it must be of type *list*). If it is a cons, the *car* (i.e., the first component) is returned; otherwise, if it is **nil**, **nil** is returned.

[*Function*]

cdr *list* => *rest-element*

This function returns the rest (i.e., the *cdr* -- all but the first element) of *list*.

list must be either a cons or **nil** (i.e., it must be of type *list*). If it is a cons, the *cdr* (i.e., the second component) is returned; otherwise, if it is **nil**, **nil** is returned.

[*Function*]

caar *list* => *object*

This function is equivalent to

(car (car *list***))**.

[*Function*]

cadr *list* => *object*

This function returns the second element of *list*. It is equivalent to

(car (cdr *list***))**.

[*Function*]

cdar *list* **=>** *object*

This function is equivalent to

(cdr (car *list***)).**

[*Function*]

cddr *list* **=>** *object*

This function is equivalent to

(cdr (cdr *list***)).**

[*Function*]

caaar *list* **=>** *object*

This function is equivalent to

(car (car (car *list***))).**

[*Function*]

caadr *list* **=>** *object*

This function is equivalent to

(car (car (cdr *list***))).**

[*Function*]

cadar *list* => *object*

This function is equivalent to

(car (cdr (car *list***))).**

[*Function*]

caddr *list* => *object*

This function returns the third element of *list*. It is equivalent to

(car (cdr (cdr *list***))).**

[*Function*]

cdaar *list* => *object*

This function is equivalent to

(cdr (car (car *list***))).**

[*Function*]

cdadr *list* => *object*

This function is equivalent to

(cdr (car (cdr *list***))).**

[*Function*]

cddar *list* => *object*

This function is equivalent to

(cdr (cdr (car *list*))).

[*Function*]

cdddr *list* => *object*

This function is equivalent to

(cdr (cdr (cdr *list*))).

[*Function*]

cons *object1 object2* => *cons*

This function creates and returns a **cons** object whose *car* and *cdr* are *object1* and *object2*, respectively.

If *object2* is a **list** object, one may think of **cons** as adding *object1* to the front of the list.

[*Function*]

ncons *object* => *cons*

This function creates and returns a **cons** object whose *car* is *object* and whose *cdr* is **nil**.

(ncons *object*) <=> **(list** *object*)

15.2 Lists

[*Function*]

endp *list* => *boolean*

This function is a predicate which is true if *list* is the object **nil**.

list must be an object of type *list*.

Implementation note: An error is signalled if the argument to **endp** is not of type **list**.

[*Function*]

list-length *list* => *length*

This function returns either an integer which represents the number of elements in *list* or **nil** if *list* is circular.

The argument must be of type **list**.

[*Function*]

nth *n list* => *object*

This function returns the *n*th element of *list*, where the first element of *list* is the 0th element.

[*Function*]

first *list* => *element*

This function returns the first *element* of *list*. It is equivalent to **car**.

[*Function*]

second *list* => *element*

This function returns the second *element* of *list*. It is equivalent to **cadr**.

[*Function*]

third *list* => *element*

This function returns the third *element* of *list*. It is equivalent to **caddr**.

[*Function*]

rest *list* => *rest-list*

This function returns the rest of *list* (i.e., the list containing the 2nd through the last element). It is equivalent to **cdr**.

[*Function*]

nthcdr *n list* => *sub-list*

This function returns the *n*th successive **cdr** of *list*. In other words it returns the sublist of *list* containing the *n*th+1 through the last elements. Note that the 0th **cdr** of a list is the list itself.

[*Function*]

last *list* => *last-cons*

This function returns the last **cons** (not the last element) of *list*. Note that the last **cons** of **nil** is **nil**.

[*Function*]

list **&rest** *objects* => *list*

This function creates and returns a *list* containing all of its arguments. Given no arguments, **list** returns **nil**.

[*Function*]

list* *object* **&rest** *other-objects* => *list*

This function returns a *list* which is created by successively *cons*ing, from right to left, all but the last argument onto the last argument.

In other words, the last argument is used as the **cdr** of the last cons of the list constructed from all the other arguments. This implies that if the last argument to **list*** is a non-**nil** atom, then the *list* returned is a **dotted-list**.

[*Function*]

make-list *size* **&key** **:initial-element** => *list*

This function creates and returns a list of length *size*, all of whose elements are the **:initial-element** (which defaults to **nil**). *size* must be a non-negative integer.

[*Function*]

append **&rest** *lists* => *list*

This function concatenates the *lists* together. All but the last argument to **append** must be a **list**; the last argument may be any type of object.

append does not modify any of its arguments. It copies the top-level list structure of each argument (except the last), replacing the **cdr** of each argument's **last cons** with the argument to the right.

[*Function*]

copy-list *list* => *list-copy*

This function returns a copy of *list*. The copy is **equal** to *list* but not **eq**.

The elements of the copy are **eq** to their corresponding elements in *list* (i.e., only the top-level list structure of *list* is copied).

list may be a dotted-list, in which case the **cdr** of the **last cons** of the copy will be **eq** to the **cdr** of the **last cons** of *list*.

[*Function*]

copy-alist *a-list* => *new-alist*

This function creates and returns a copy of *a-list* in which each element of type **cons** is replaced by a new cons with the the same *first* and *rest*.

[*Function*]

copy-tree *object* => *object-copy*

This function recursively copies every cons in *object* and returns the new copy.

[*Function*]

nconc **&rest** *lists* => *list*

This function concatenates all of its arguments and returns the resulting *list*.

All of the arguments must be *lists*. The **cdr** of the **last** **cons** of each non-nil argument is replaced by the first non-nil argument to its right. The first **non-nil** argument is returned.

[*Macro*]

push *object place* => *result*

This macro replaces the list stored in the generalized variable *place* with a list created by *cons*ing *object* onto the original list.

place must be a form acceptable as a generalized variable to **setf**. *object* may be an object of any type.

If the list stored in *place* is thought of as a *push-down stack*, then **push** pushes *object* onto that stack.

Compatibility note: The value returned by **push** is undefined.

[*Macro*]

pushnew *object place* => *result*

This macro replaces the list stored in the generalized variable *place* with a list created by *adjoin*ing *object* onto the original list.

place must be a form acceptable as a generalized variable to **setf**. *object* may be an object of any type. **adjoin** conses an object onto a list if and only if, the object is not already a **member** of that list.

If the list stored in *place* is thought of as a set, then **pushnew** adds *object* to that set.

Compatibility note: The value returned by **pushnew** is undefined.

[*Macro*]

pop *place* => *object*

This macro replaces the list stored in the generalized variable *place* with the **cdr** of that list and returns the **car** of that list.

place must be a form acceptable to **setf** as a generalized variable. The object stored at *place* must be a list.

If the list stored at *place* is thought of as a *push-down stack*, then **pop** *pops* the top element from the stack and returns it.

[*Function*]

butlast *list* **&optional** *n* => *truncated-list*

This function creates and returns a list containing all but the last *n* elements of *list*.

n must be a non-negative integer. The argument *list* is not modified in any way. If *list* has fewer than *n* elements, the empty list '()' is returned.

[*Function*]

nbutlast *list* **&optional** *n* => *truncated-list*

This function returns a list containing all but the last *n* elements of *list*. *list* may be modified in the process.

n must be a non-negative integer. If *list* contains *n* or fewer elements, the empty list '()' is returned and *list* is left unmodified. On the other hand, if *list* contains more than *n* elements, **nbutlast** replaces the **cdr** of the cons N+1 from the end of *list* with **nil** and returns the modified *list*.

[*Function*]

ldiff *list sublist* => *new-list*

This function creates and returns a list containing those elements of *list* that appear before *sublist*.

Both *list* and *sublist* must be lists. If one of the conses which make up *list* has a **cdr** containing *sublist*, then the copy returned by **ldiff** will end with that **cons** (i.e., the **cdr** of that **cons** will be **nil** instead of *sublist*). Otherwise, a complete copy of *list* is returned (i.e., the copy will be **equal** to *list*). *list* is not modified in any way.

ldiff may be thought of as returning the *difference* of two lists.

15.3 Alteration of List Structure

[*Function*]

rplaca *cons object* => *cons*

This function replaces the **car** of *cons* with *object* and returns (the modified) *cons*.

cons must be an object of type **cons**. *object* may be an object of any type.

rplaca stands for *RePLAce CAr* and is pronounced *replacuh*.

[*Function*]

rplacd *cons object* => *cons*

This function replaces the **cdr** of *cons* with *object* and returns (the modified) *cons*.

cons must be an object of type **cons**. *object* may be an object of any type.

rplacd stands for *RePLAce CDr* and is pronounced *replacduh*.

[*Function*]

rplacb *cons1 cons2* => *cons1*

This function replaces the **car** and **cdr** of *cons1* with the **car** and **cdr** of *cons2*, respectively, and returns (the modified) *cons1*.

cons1 and *cons2* must be of type **cons**.

rplacb stands for *RePLAce Both the car and cdr* and is pronounced *replacbuh*.

[*Function*]

snoc *cons object* => *list*

This function replaces the **cdr** of *cons* with the **ncons** of *object*.

15.4 Substitution of Expressions

[*Function*]

subst *new old tree* => *new-tree*

This function returns a tree with *new* substituted for every occurence of *old*. The original *tree* is not modified in any way.

The three arguments to **subst** may be objects of any **type**. If *old* is **eql** to *tree*, then **subst** returns *new*. If *tree* is not of type **cons** and is not **eql** to *old*, then **subst** returns *tree*. Otherwise, *tree* is a **cons** and **subst** is recursively applied to its **car** and **cdr**. **subst** returns a **cons** containing the two returned trees. Note that this returned **cons** may be the original **cons** only if the two returned trees are **eql** to their respective originals.

This definition implies that if no substitution is made or if *old* is **eql** to *new*, the original *tree* may be returned. Otherwise, a new tree (parts of which will be **eql** to the original tree) must be returned.

Compatibility note: Keyword arguments are not supported.

[*Function*]

sublis *a-list tree* => *new-tree*

This function performs the substitutions specified by *a-list* upon *tree* and returns the resulting tree. The original *tree* is not modified in any way.

a-list must be an association list, while *tree* may be an object of any type. If (**assoc** *tree a-list*) returns a cons (i.e., is true), then **subst** returns the **cdr** of that **cons**; otherwise, if *tree* is not of type **cons**, then **subst** returns *tree*. Otherwise, *tree* is a **cons** and **subst** is recursively applied to its **car** and **cdr**. **subst** returns a **cons** containing the two returned trees. Note that this returned **cons** may be the original **cons** only if the two returned trees are **eql** to their respective originals.

In effect, **sublis** performs several **subst** operations at once.

Compatibility note: Keyword arguments are not supported.

15.5 Using Lists as Sets

[*Function*]

member *item list* **&key :test** => *list-boolean*

This function is a predicate which is true if and only if *list* contains an element which satisfies the **:test**, i.e., (**funcall** *test item element*) is true. If *member* is true, it returns the tail of *list* beginning with the first element satisfying **:test**.

:test defaults to **eql**.

Compatibility note: Only the **:test** keyword argument is supported.

[*Function*]

member-if *test list* => *nil/list-tail*

This function is a predicate which is true if and only if *list* contains an element which satisfies *test*, i.e., **(funcall** *test element***)** is true. If *member* is true, it returns the tail of *list* beginning with the first element satisfying *test*.

Compatibility note: Keyword arguments are not currently supported.

[*Function*]

tailp *sublist list* => *boolean*

This function is a predicate which is true if and only if

(nthcdr *n list***)** => *sublist*

for some *n* (0 <= *n* <= **(length** *list***)**).

In other words, *sublist* must be either **nil** or one of the conses which make up *list*.

[*Function*]

adjoin *item list* => *new-list*

This function adds *item* to *list* (using **cons**) and returns the resulting list only if *item* is not already a **member** of *list*; otherwise, the original *list* is returned.

item may be an object of any type, while *list* must be of type list. The original *list* is not modified in any way. If one thinks of *list* as representing a set, then **adjoin** may be thought of as adding a new item to the set.

```
(adjoin item list)
    <=> (if (member item list)
            list
            (cons item list))
```

Compatibility note: No keyword arguments are supported.

15.6 Association Lists

An *association list* (or *a-list* for short) is a list whose elements are either **nil** or dotted-pairs (i.e., conses). An a-list is used to represent a mapping.

[*Function*]

acons *key datum a-list* => *new-a-list*

This function creates and returns a new association list by adding the association pair, *key* and *datum*, to the front of the argument, *a-list*.

```
(acons key datum a-list)
    <=> (cons (cons key datum) a-list)
```

[*Function*]

pairlis *key-list datum-list* **&optional** *a-list* => *new-a-list*

This function returns an association list formed by adding the association pairs created by pairing each *key* element in *key-list* with its corresponding *datum* in *datum-list* to the front of the optional *a-list*.

key-list and *datum-list* must be lists of equal length. *a-list* (which defaults to **nil**) must also be a list. The order in which the association pairs are added (i.e., consed) to *a-list* is undefined.

[*Function*]

assoc *item a-list* **&key :test** => *assoc-pair*

This function returns either the first association pair (i.e., a **cons**) contained in *a-list* whose key (i.e., **car**) satisfies the **:test**, or **nil** if no such pair exists.

item may be an object of any type, while *a-list* must be a list all of whose elements are lists. **:test** (which defaults to **eql**) must be a functional predicate. A key *satisfies* the **:test** if and only if

(funcall test *item key***)**

is true.

assoc ignores **nil** within the *a-list* being searched.

Compatibility note: Only the **:test** keyword argument is supported.

[*Function*]

rassoc *item a-list* **&key :test** => *assoc-pair*

This function returns either the first association pair (i.e., a **cons**) contained in *a-list* whose datum (i.e., **cdr**) satisfies the **:test**, or **nil** if no such pair exists.

item may be an object of any type, while *a-list* must be a list all of whose elements are lists. **:test** (which defaults to **eql**) must be a functional predicate. A datum *satisfies* the **:test** if and only if

(funcall *test item datum***)**

is true.

assoc ignores any **nil** within the *a-list* being searched.

Compatibility note: Only the **:test** keyword argument is supported.

Chapter 16

Hash Tables

16.1 Hash Table Functions

Currently, GCLISP does not support any hash table functions.

16.2 Primitive Hash Function

[*Function*]

sxhash *object* => *hash-code*

This function is a *hash function*. Given an *object* of any type, this function returns a non-negative integer (called the *hash code* of *object*).

sxhash hashes on tree structure (also called, *hashing on* **equal**). This means that

(equal *x y***) implies (= (sxhash** *x***) (sxhash** *y***))**

in other words, **sxhash** takes the entire tree structure of *object* into account when generating its hash code.

Chapter 17

Arrays

17.1 Array Creation

[*Function*]

make-array *dimension* **&key :element-type**
:initial-element :initial-contents :fill-pointer
:leader-length :named-structure-symbol
=> *vector*

This function creates and returns a one-dimensional **array** (also called a **vector**).

dimension must be a non-negative integer. It specifies the length of the *vector*.

:element-type must be one of the following type specifiers: **t** (the default), **string-char**, or **(unsigned-byte 8)**. It specifies what type of element may be stored in the *vector*. Note that the type specifier **t** allows all types.

:initial-element must be an object of the type specified by **:element-type**. If provided, each element of the created *vector* is initialized to it.

:initial-contents must be a list whose length is equal to *dimension*. The elements in the list must be of the type specified by **:element-type**. The *n*th element of the *vector* is initialized to the *n*th element of the list.

:fill-pointer must be either **t**, **nil** (the default), or a non-negative integer less than or equal to *dimension*. If **:fill-pointer** is **nil**, then *vector* will not have a fill pointer; otherwise *vector* will have a fill pointer which is initialized to either the end of the *vector* (by specifying **t**) or some particular offset (by specifying an integer).

:leader-length must be a non-negative integer. If it is positive, *vector* will have an array leader of that length.

:named-structure-symbol must be a symbol. The object returned by **make-array** will be of type **structure**. The symbol is made the name of the structure.

:initial-element and **:initial-contents** may not both be specified. If neither is specified, the initial values of the *vector* elements are undefined.

Compatibility note: Multi-dimensional arrays are not supported. Not all keyword arguments are supported. bit-vectors are not supported. **:initial-contents** must be a list. Array-based structures can be created.

[*Function*]

vector **&rest** *objects* => *simple-general-vector*

This function creates and returns a simple general vector whose initial contents are *objects*.

(vector *obj1 obj2 ... objn***)**
<=> (make-array *n*
:initial-contents
(list *obj1 obj2 ... objn***))**

17.2 Array Access

[*Function*]

aref *vector index* => *array-element*

This function returns the value of the element at position *index* in *vector*.

vector must be of type **vector** (i.e., a one dimensional array). *index* must be a non-negative integer less than the dimension of *vector*. Note that vectors are indexed from zero.

aref can access any element in *vector* regardless of the value of a fill pointer for the *vector* (if one exists).

17.3 Array Information

[*Function*]

array-in-bounds-p *vector* **&rest** *index* => *boolean*

This function is a predicate which is true if and only if *index* is greater than 0 and less than the length of *vector*.

vector must be a vector, and *index* must be an integer.

[*Function*]

array-active-length *array* => *length*

This function returns the fill pointer of *array* if it has one; otherwise, it returns the total number of elements in *array*.

[*Function*]

array-length *array* => *length*

This function returns the total number of elements in *array*, regardless of the fill pointer.

17.4 Functions on Arrays of Bits

GCLISP does not currently support arrays of bits.

17.5 Fill Pointers

[*Function*]

array-has-fill-pointer-p *array* => *boolean*

This function is a predicate which is true if and only if *array* has a fill pointer.

array must be an array.

[*Function*]

fill-pointer *vector* => *integer*

This function returns the fill pointer of *vector*.

vector must be of type **vector** and must have a fill pointer. The fill pointer of a vector is always a non-negative integer less than the length of the vector.

Implementation note: An error is signalled if the argument to **fill-pointer** is not a vector with a fill pointer.

[*Function*]

vector-push *new-element vector* => *previous-active-length*

This function attempts to extend the *active* length of *vector*, storing *new-element* into the new active element and returning the previous active length.

vector must be of type **vector** and must have a fill pointer. *new-element* must be of the type specified by *vector*'s **:element-type**.

If fill pointer is equal to the length of *vector*, *vector* is left unmodified, and **nil** is returned; otherwise, *new-element* is stored at the position indicated by the fill pointer, fill pointer is incremented by one, and the index where *new-element* was stored is returned.

[*Function*]

vector-pop *vector* => *element*

This function decreases the *active* length of *vector* by one and returns the value of the element designated by new value of the fill pointer.

The argument to **vector-pop** must be of type **vector** and it must have a fill pointer.

If the fill pointer is zero, an error is signalled; otherwise, the fill pointer is decremented by one, and the value of the element at the position specified by fill pointer is returned.

17.6 Changing the Dimensions of an Array

GCLISP does not currently support adjustable arrays.

17.7 Array Leaders

[*Function*]

array-has-leader-p *array* => *boolean*

This function is a predicate which is true if and only if *array* has an array leader.

[*Function*]

array-leader *array-with-leader index* => *object*

This function returns the *index*th element of *array-with-leader*'s array leader.

[*Function*]

array-leader-length *array* => *length*

This function returns the length of *array*'s array leader if it has one and **nil** otherwise.

[*Function*]

store-array-leader *object array-with-leader index*
=> *object*

This function stores *object* in the *index*th position of *array-with-leader*'s array leader.

object is returned.

17.8 Copying the Contents of an Array

[*Function*]

copy-array-contents *from-array to-array* => *t*

This function copies the contents of the *from-array* to the *to-array*.

If *from-array* has more elements than *to-array*, the excess *from-array* elements are ignored. If *to-array* has more elements than *from-array*, its excess elements are filled with **nil** (if it is a general array), or zero (if it is a string-char or (unsigned-byte 8) array).

A fill pointer in either array is ignored.

Chapter 18

Strings

18.1 String Access

[*Function*]

char *string index* **=>** *character*

This function returns the *index*th character of *string*.

18.2 String Comparison

[*Function*]

string= *string1 string2* **&key :start1**
    :end1 :start2 :end2
    => *boolean*

This function is a predicate which is true if and only if the specified characters of *string1* are equal to their corresponding characters in *string2*. Strings of unequal lengths are not equal.

string1 and *string2* must both be of type **string.** The keyword arguments **:start1** and **:end1** (whose values must be non-negative integers), specify the range of positions in *string1* to be included in the comparison. The range has an inclusive lower (**:start1**) bound and an exclusive upper (**:end1**) bound. The keyword arguments **:start2** and **:end2** are defined analogously for *string2*.

Compatibility note: *string1* and *string2* cannot be symbols.

[*Function*]

string-equal *string1 string2* **&key :start1 :end1 :start2 :end2** => *boolean*

This function is a predicate which is true if and only if the specified characters of *string1* are **char-equal** (i.e., equal ignoring differences in case) to their corresponding characters in *string2*. Strings of unequal lengths are not equal.

string1 and *string2* must both be of type **string**. The keyword arguments **:start1** and **:end1** (whose values must be non-negative integers), specify the range of positions in *string1* to be included in the comparison. The range has an inclusive lower (**:start1**) bound and an exclusive upper (**:end1**) bound. The keyword arguments **:start2** and **:end2** are defined analogously for *string2*.

Compatibility note: *string1* and *string2* cannot be symbols.

[*Function*]

string< *string1 string2* **&key :start1 :end1 :start2 :end2** => *nil/index*

This function is a predicate which is true if and only if *string1* is lexicographically less than *string2*.

string1 and *string2* must both be of type **string**. The keyword arguments **:start1** and **:end1** (whose values must be non-negative integers), specify the range of positions in *string1* to be included in the comparison. The range has an inclusive lower (**:start1**) bound and an exclusive upper (**:end1**) bound. The keyword arguments **:start2** and **:end2** are defined analogously for *string2*.

Compatibility note: *string1* and *string2* cannot be symbols.

[*Function*]

string-lessp *string1 string2* **&key :start1 :end1 :start2 :end2** => *index-boolean*

This function is a predicate which is true if and only if *string1* is lexicographically less than *string2*, ignoring differences in case.

string1 and *string2* must both be of type **string**. The keyword arguments **:start1** and **:end1** (whose values must be non-negative integers), specify the range of positions in *string1* to be included in the comparison. The range has an inclusive lower (**:start1**) bound and an exclusive upper (**:end1**) bound. The keyword arguments **:start2** and **:end2** are defined analogously for *string2*.

Compatibility note: *string1* and *string2* cannot be symbols.

[*Function*]

string-search *key string* **&optional** *from to* => *nil/index*

This function searches (in a case-sensitive manner) for the string *key* in the string *string*, returning a non-**nil** value if it is found and **nil** otherwise.

Both *key* and *string* must be strings. *from* is an integer that specifies the position within *string* to begin the search; it defaults to 0. *to* is an integer that specifies the position (exclusive) to end the search; it defaults to the length of *string*.

If an instance of *key* is found within *string*, the index of the first character of that instance is returned.

Note: The empty string ("") is a substring of every string.

[*Function*]

string-search* *key string* **&optional** *from to* => *nil/index*

This function searches (without regard to case) for the string *key* in the string *string*, returning a non-nil value if it is found and **nil** otherwise.

Both *key* and *string* must be strings. *from* is an integer that specifies the position within *string* to begin the search; it defaults to 0. *to* is an integer that specifies the position (exclusive) to end the search; it defaults to the length of *string*.

If an instance of *key* is found within *string*, the index of the first character of that instance is returned.

Note: The empty string ("") is a substring of every string.

18.3 String Construction and Manipulation

[*Function*]

string-append **&rest** *strings* => *concatenated-string*

This function concatenates copies of *strings* into a single string.

[*Function*]

string-left-trim *character-bag string* => *trimmed-string*

This function returns a substring of *string* beginning with the first character of *string* which is not contained in *character-bag*.

Both arguments to **string-left-trim** must be of type string. The substring returned by **string-left-trim** is not a displaced array.

Implementation note: If no characters are *trimmed*, *string* itself is returned.

Compatibility note: Both arguments must be strings.

[*Function*]

string-right-trim *character-bag string* => *trimmed-string*

This function returns a substring of *string* ending with the first character of *string* which is not contained in *character-bag*.

Both arguments to **string-right-trim** must be of type string. The substring returned by **string-right-trim** is not a displaced array.

Implementation note: If no characters are *trimmed*, *string* itself is returned.

Compatibility note: Both arguments must be strings.

[*Function*]

string *object* => *string*

This function returns the string-type equivalent of *object*. If *object* is of type **string**, it is returned. If *object* is of type **symbol**, its print name is returned. If *object* is a string character, a string containing that one character is returned. If *object* is not one of the above types, an error is signalled.

Chapter 19

Structures

19.1 Introduction to Structures

The GCLISP structure facility conforms to the COMMON LISP standard except that only the the following **defstruct** options are currently supported:

- **:conc-name**
- **:constructor**
- **:predicate**
- **:print-function**
- **:type**
- **:named**
- **:initial-offset**

19.2 How to Use Defstruct

[*Macro*]

defstruct {*Name* | (*Name* {*Option*}*)}
    {*Slot-description*}+ => *name*

This macro defines a structured data type.

The following options are supported: **:conc-name**, **:constructor**, **:predicate**, **:print-function**, **:type**, **:named**, **:initial-offset**.

19.3 Using the Automatically Defined Constructor Function

See the COMMON LISP Reference Manual.

19.4 Defstruct Slot-Options

Not currently supported.

19.5 Defstruct Options

See the COMMON LISP Reference Manual.

19.6 By-position Constructor Functions

Not currently supported.

19.7 Structures of Explicitly Specified Representational Type

See the COMMON LISP Reference Manual.

19.7.1 Unnamed Structures

See the COMMON LISP Reference Manual.

19.7.2 Named Structures

[*Function*]

named-structure-p *object* **=>** *nil/name*

This function returns **nil** if *object* is not a named structure; otherwise, if *object* is a named structure, its *name* is returned.

[*Function*]

named-structure-symbol *named-structure* **=>** *name-symbol*

This function returns the *named-structure*'s name (a symbol).

19.7.3 Other Aspects of Explicitly Specified Structures

See the COMMON LISP Reference Manual.

Chapter 20

The Evaluator

20.1 Run-Time Evaluation of Forms

[*Function*]

eval *form* => *object*

This function is *The Evaluator*. It *evaluates form* and returns the result of that evaluation.

form must be a valid (i.e. meaningful) form. Note that in the evaluation of an **eval** function call form, the argument form is evaluated twice: once because it is an argument to a function, and once because that function is the evaluator.

[*Variable*]

evalhook => *eval-hook-function*

The value of this variable is used to replace **eval** in the evaluation of forms.

If the value of this variable is **nil** (the default), **eval** is used to evaluate forms. If the value of this variable is not **nil**, then it must be a function (call it *eval-func*) of one argument.

When a form is to be evaluated, this *eval-func* is called with the form as an argument. The value returned by *eval-func* is used as the value of the form.

During the evaluation of *eval-func*, the two variables ***evalhook*** and ***applyhook*** are bound to **nil**.

If a throw to a listener loop occurs, the same two variables are set to **nil**.

Implementation note: If a *break* occurs, the hook variables are bound to **nil** within the *break*.

Compatibility note: The eval hook function does not take an environment argument.

[*Variable*]

applyhook => *apply-hook-function*

The value of this variable is used to replace **apply** in the application of functions to arguments.

If the value of this variable is **nil** (the default), **eval** uses **apply** to apply a function to its arguments. If the value of this variable is not **nil**, then it must be a function (call it *apply-func*) of two arguments.

When **eval** is about to apply a function to its list of arguments, this *apply-func* is called (instead of **apply**) with the function as the first argument and the argument list as the second. The value returned by *apply-func* is used as the value of the function call form.

During the evaluation of *apply-func*, the two *hook* variables ***evalhook*** and ***applyhook*** are bound to **nil**.

If a throw to a listener loop occurs, the two hook variables are set to **nil**.

Implementation note: If a *break* occurs, the hook variables are bound to **nil** within the *break*.

Compatibility note: The apply hook function does not take an environment argument. Also, the apply hook function is called when special forms are evaluated.

[*Function*]

evalhook *form evalhookfn applyhookfn* => *values*

This function binds ***evalhook*** to *evalhookfn* and ***applyhook*** to *applyhookfn* after beginning the evaluation of *form* but before any subsidiary evaluations (e.g., for arguments in *form*) are begun.

form must be a valid form. *evalhookfn* and *applyhookfn* must both be functions. **evalhook** returns the results of evaluating form.

Compatibility note: evalhook does not take an environment argument.

[*Function*]

applyhook *function args evalhookfn applyhookfn* => *values*

This function binds ***evalhook*** to *evalhookfn* and ***applyhook*** to *applyhookfn* after applying *function* to *args*, but before any subsidary evaluations (e.g. within the body of *function*) are begun.

function and *args* must be acceptable as arguments to **apply**. *evalhookfn* and *applyhookfn* must both be functions. **applyhook** returns the results of applying *function* to *args*.

Compatibility note: applyhook does not take an environment argument.

20.2 The Top-Level Loop

In GCLISP, the Top-Level Loop is merely the top-most invocation of the function **listener**.

[*Function*]

listener &optional *herald-string*

This function invokes a read-eval-print loop.

herald-string is bound to the global variable ***listener-name*** and is printed when the read-eval-print loop is first entered and when **listener** catches a *throw* to the tag **:listener**.

listener never returns a value since the read-eval-print loop is an infinite-loop.

[*Variable*]

listener-name => *string*

The value of this global variable is a string that is printed when the **listener** read-eval-print loop is first entered and when **listener** catches a *throw* to the tag **:listener.**

[*Variable*]

+ => *form*

The value of this variable is the second most recently read top-level form. In other words, during the current Top-Level interaction, **+** is bound to the form read by the previous Top-Level interaction. Before a new interaction begins, **+** is assigned the current value of **-.**

[*Variable*]

++ => *form*

The value of this variable is the third most recently read top-level form. In other words, during the current Top-Level interaction, **++** is bound to the top-level form read two interactions ago. Before a new interaction begins, **++** is assigned the current value of **+.**

[*Variable*]

+++ => *form*

The value of this variable is the fourth most recently read top-level form. In other words, during the current Top-Level interaction, **+++** is bound to the top-level form read three interactions ago. Before a new interaction begins, **+++** is assigned the current value of **++.**

[*Variable*]

- => *form*

The value of this variable is the most recently read top-level form. Each time a form is read by the top-level loop, it is assigned to -.

[*Variable*]

***** => *object*

The value of this variable is the first result returned by the most recently evaluated top-level form. In other words, during the current Top-Level interaction, * is bound to the (first) result of the last interaction. Each time a top-level form is evaluated by the top-level loop, the first result is assigned to *.

[*Variable*]

****** => *object*

The value of this variable is the first result returned by the second most recently evaluated top-level form. In other words, during the current Top-Level interaction, ** is bound to the (first) result of the second to last interaction. Before a new interaction begins, ** is assigned the value of *.

[*Variable*]

******* => *object*

The value of this variable is the first result returned by the third most recently evaluated top-level form. In other words, during the current Top-Level interaction, *** is bound to the (first) result of the third to last interaction. Before a new interaction begins, *** is assigned the value of **.

[*Variable*]

/ => *object*

The value of this variable is a list of the results returned by the most recently evaluated top-level form. In other words, during the current Top-Level interaction, / is bound to the results of the last interaction. Each time a top-level form is evaluated by the top-level loop, a list of the results is assigned to /.

[*Variable*]

// => *object*

The value of this variable is a list of the results returned by the second most recently evaluated top-level form. In other words, during the current Top-Level interaction, // is bound to a list of the results of the second to last interaction. Before a new interaction begins, // is assigned the value of /.

[*Variable*]

/// => *object*

The value of this variable is a list of the results returned by the third most recently evaluated top-level form. In other words, during the current Top-Level interaction, /// is bound to a list of the results of the third to last interaction. Before a new interaction begins, /// is assigned the value of //.

[*Variable*]

prompt => *function*

The global value of this variable is a function which is called each time the input-editor performs a *refresh*.

The function must take no arguments. The values it returns are discarded. The intended purpose of the function is to print a prompt on the ***standard-output*** stream. The function may assume that its output will be printed on a fresh line.

The initial value of ***prompt*** is a function which prints the string "* " on the ***standard-output*** stream. (Unless the current package is not *user*, in which case it prints the name of the current package followed by ": " as a prompt.)

Chapter 21

Streams

21.1 Standard Streams

[*Variable*]

standard-input => *input-stream*

The initial global value of this variable is an input stream. By default, the LISP Reader reads from the input stream which is assigned (or bound) to this variable.

[*Variable*]

standard-output => *output-stream*

The initial global value of this variable is an output stream. By default, the LISP Printer writes to the output stream which is assigned (or bound) to this variable.

[*Variable*]

error-output => *output-stream*

The initial global value of this variable is an output stream.

Compatibility note: Currently, none of the error system functions use this stream for output. They use *debug-io* instead.

[*Variable*]

query-io => *input/output-stream*

The initial global value of this variable is an input/output stream. The functions **y-or-n-p** and **yes-or-no-p** use the stream that is the value of this variable. This stream should be used for querying the user.

[*Variable*]

debug-io => *input/output-stream*

The initial global value of this variable is an input/output stream that is used for interactive debugging purposes.

Compatibility note: The error system functions (e.g., **error** and **cerror**) use this stream instead of ***error-output***.

[*Variable*]

terminal-io => *input/output-stream*

The initial global value of this variable is an input/output stream that connects to the user's console. Normally, writing to this stream causes the output to appear on the console display, while reading reads the characters typed at the console keyboard.

The value of this variable should not be changed.

[*Variable*]

trace-output => *output-stream*

The initial global value of this variable is an output stream. The function **trace** writes to the output stream which is assigned (or bound) to this variable.

21.2 Creating New Streams

[*Function*]

make-synonym-stream *symbol* => *stream*

This function creates and returns a *synonym* stream.

Whenever an operation is performed on this stream (call it *a*), *symbol* must be bound to some stream (call it *b*). Any operation performed on *a* will actually be performed on *b*.

[*Function*]

make-string-input-stream *string* **&optional** *start end* => *input-stream*

This function creates and returns an input stream which will produce the characters contained in the substring (delimited by *start* and *end*) of *string*.

[*Function*]

make-string-output-stream => *string-output-stream*

This function returns an output stream that will accumulate all output written to it. The accumulated output may be retrieved using get-output-stream-string.

[*Function*]

get-output-stream-string *string-output-stream* => *string*

This function returns a string containing all the characters so far accumulated by *string-output-stream*, resetting the stream to zero accumulated characters.

string-output-stream must be a **stream** produced by **make-string-output-stream.**

21.3 Operations on Streams

[*Function*]

close *stream* => *nil*

This function closes *stream*. A closed stream may not be read from or written to.

Compatibility note: The **:abort** argument is not currently supported.

[*Function*]

close-all-files => *list*

This function closes all open streams that are connected to files and returns a list of all the previously open files.

21.4 Using Streams as Functions

In GCLISP, streams are a type of function. Thus, besides acting as arguments to functions such as **read** and **print**, streams may be applied to arguments using **funcall** or **apply**.

When a stream is applied to some arguments, the first argument must always be a keyword symbol. This keyword indicates the *operation* that the stream is to perform using the rest of the arguments. For this reason, the keyword is often called the *operation*. For example, the function call,

```
(funcall *terminal-io* :write-char #\A)
```

will perform the *write-char* operation (with the letter 'A' as its argument) on the stream connected to the terminal. (Note that this function call is equivalent to the function call **(write-char #\A *terminal-io*).**)

The above function call also has the flavor of a message (no pun intended): **funcall** acts as the message sending mechanism, ***terminal-io*** acts as the object receiving the message, **:write-char** acts as the message name, **#\A** acts as a message argument, and the value returned by the function call acts as the object returned by the receiver object. In order to encourage the message-passing metaphor, GCLISP defines the function **send**.

[*Function*]

send *function* **&rest** *arguments* => *function-call-results*

This function calls *function* with *arguments* and returns the results of this function call.

send is identical to **funcall**, but connotes the message-passing metaphor to the user.

Input streams must support the following two basic operations:

:read-char => *nil/character*
: Inputs the next available character from the stream. If there is no character available, it is waited for. If the end-of-file is reached, **nil** is returned.

:unread-char *character* => *character*
: Pushes *character* (which must be the most recently read character) back into the input stream. This makes *character* the next available character. The **:unread-char** operation cannot be repeated unless a character has been read (e.g., using **:read-char**) since the previous **:unread-char** operation.

Output streams need only support one basic operation:

:write-char *character* => *character*
: Outputs *character* on the stream and returns the character written.

These operations, although related to the functions **read-char**, **unread-char** and **write-char** respectively, do not allow the optional arguments that their corresponding functions allow.

All streams must support the following operation:

:which-operations => *operations-list*
Returns a list of keywords, each of which names an operation that is explicitly supported by this stream.

Most of the other stream operations can built up from these basic operations using the **stream-default-handler**, described below.

21.5 User Written Streams

Since streams are merely a type of function, it is possible for users to define functions that can be used as streams.

A user-written input stream function must handle at least three operations: **:read-char**, **:unread-char**, and **:which-operations**. An example of a very simple user-written input stream is the following:

```
(defun newline-input-stream (operation &optional ignore)
   (case operation
      (:read-char #\Newline)
      (:unread-char)
      (:which-operations
         '(:read-char :unread-char :which-operations))
      (otherwise
         (error "Unknown input stream operation: ~S"
            operation))))
```

This stream produces an infinite number of Newlines.

A user-written output stream must handle two operations: **:write-char** and **:which-operations**. The following is a very simple example of a user-written output stream:

```
(defparameter *list* '())

(defun list-output-stream (operation &optional arg)
   (case operation
      (:write-char
         (setf *list*
```

```
          (append *list* (list arg))))
      (:which-operations
         '(:write-char :which-operations))
      (otherwise
         (error "Unknown input stream operation: ~S"
           operation))))
```

This output stream collects the actual characters output into a list, which is the value of the global variable ***list***.

A simple method of extending the number of operations that a user-written stream may handle is to use the function **stream-default-handler.**

[*Function*]

stream-default-handler *stream operation* **&rest** *arguments* => *operation-result*

This function attempts to handle *operation* on *stream*, given *arguments*.

It is normally called by a user-written stream that has been called with an operation that the user-written stream does not explicity handle. In such a case the user-written stream merely passes on the operation and its arguments to **stream-default-handler.**

21.6 Window Streams

See Appendix C.

Chapter 22

Input/Output

22.1 Printed Representation of Lisp Objects

22.1.1 What the Read Function Accepts

22.1.2 Parsing of Numbers and Symbols

[*Variable*]

read-base => *integer*

The value of this variable determines the radix in which integers will be read. The integer may be between 2 and 36 (inclusive). The initial value of this variable is 10.

22.1.3 Macro Characters

22.1.4 Standard Dispatching Macro Character Syntax

22.1.5 The Readtable

[*Function*]

set-syntax-from-char *to-char from-char* => *to-char*

This function copies the readtable syntax information for *from-char* to *to-char* and returns *to-char*.

Only the following syntatic type information is copied: *whitespace*, *constituent*, *single escape*, *multiple escape*, or *macro*. In addition, if a macro character is copied, its macro definition function is also associated with the *to-char*.

Compatibility note: No *readtable* arguments are allowed.

[*Function*]

set-macro-character *char function* => *char*

This function affects the *readtable*, causing the LISP Reader to treat *char* as a macro character with *function* as its associated function. **set-macro-character** returns **t**.

function must be a function of two arguments. The first argument is the current input stream and the second argument is *char*. *function*'s only side-effect must be its affect on the stream.

function may return one or two values. If the second value is **nil** or if only a single value is returned, the first value is immediately returned by the LISP Reader. Otherwise, if the second value is **non-nil**, the macro character and any characters read by its associated function contribute nothing to the object being read.

Compatibility note: No optional arguments are allowed. *function* returns a **non-nil** second value to get the same effect as returning zero values.

22.1.6 What the Print Function Produces

[*Variable*]

print-escape => *boolean*

The value of this variable controls whether or not the printer includes appropriate escape characters in printed representations. If the value is **non-nil** (the initial value is **t**), escape characters will be included; otherwise, if the value is **nil**, no escape characters will be included.

All the print functions bind this variable to the appropriate value.

[*Variable*]

print-base => *integer*

The value of this variable determines the radix in which integers are printed. The integer may be between 2 and 36 (inclusive). The initial value of this variable is 10.

[*Variable*]

print-radix => *boolean*

The value of this variable controls the printing of radix specifiers. If the value is non-**nil**, all integers will be printed with a radix specifier. For example, if the current base is decimal, numbers will be printed with a trailing decimal point.

Otherwise, if the value is **nil**, no radix specifiers are printed.

[*Variable*]

print-level => *nil/integer*

The value of this variable determines the number of levels of a nested data structure that will be printed.

If the value is **nil**, every level will be printed. Otherwise, the value must be a non-negative integer.

[*Variable*]

print-length => *nil/integer*

The value of this variable determines the number of elements of a composite data structure that will be printed.

If the value is **nil**, every element will be printed. Otherwise, the value must be a non-negative integer.

22.2 Input Functions

22.2.1 Input from Character Streams

[*Function*]

read **&optional** *input-stream eof-error-p*
eof-value recursive-p => *object*

This function reads in the printed representation of a LISP object from *input-stream* and returns the corresponding LISP object (creating it if necessary).

[*Function*]

read-preserving-whitespace **&optional** *input-stream*
eof-error-p eof-value recursive-p
=> *object*

This function reads in the printed representation of a LISP object from *input-stream* and returns the corresponding LISP object (creating it if necessary).

read-preserving-whitespace is identical to **read** except that the former does not discard the delimiting whitespace character which follows an object while the latter does.

Note: If *recursive-p* is not **nil**, then **read-preserving-whitespace** behaves *exactly* like **read**.

[*Function*]

read-line **&optional** *input-stream eof-error-p*
eof-value recursive-p
=> *line-string eof-p*

This function reads in characters until it reads a **Newline** character or the end of file is encountered; it then returns two values: a string containing all the characters read except for the **Newline** and **t** or **nil** depending upon whether or not the end of file was encountered, respectively.

If the end of file is encountered before any characters are read, the following occurs: if *eof-error-p* is **nil**, *eof-value* is returned; otherwise an error will be signalled.

[*Function*]

read-char **&optional** *input-stream eof-error-p eof-value recursive-p* => *character*

This function reads one character from *input-stream*, and returns it as a character object.

[*Function*]

unread-char *character* **&optional** *input-stream* => *character*

This function puts *character*, which must be the character that was most recently produced by *input-stream*, back onto the front of *input-stream*. Thus, *character* will be the next character produced by *input-stream*.

[*Function*]

read-from-string *string* **&optional** *eof-error-p eof-value* **&key** **:start** **:end** **:preserve-whitespace** => *object first-unread-char-index*

This function reads in the printed representation of a LISP object from the substring of *string* delimited by **:start** and **:end**, returning the corresponding LISP object (creating it if necessary) and the index of the first character in *string* that was not read. If **:preserve-whitespace** is non-nil, the LISP Reader will behave as if it had been invoked with **read-preserving-whitespace**.

22.2.2 Input from Binary Streams

[*Function*]

read-byte &optional *binary-input-stream eof-error-p eof-value* => *fixnum*

This function reads one 8-bit byte from the *binary-input-stream* and returns it as a *fixnum* in the range 0 to 255 (inclusive).

Compatibility note: *binary-input-stream* may be a character stream. *binary-input-stream* is optional (it defaults to ***standard-input***).

22.3 Output Functions

22.3.1 Output to Character Streams

[*Function*]

prin1 *object* **&optional** *output-stream* => *object*

This function outputs the printed representation of *object* to *output-stream*.

The printed representation of *object* output by **prin1** includes the escape characters (\ and |) as necessary, in order that they may be read in correctly.

[*Function*]

print *object* **&optional** *output-stream* => *object*

This function outputs the printed representation of *object* to *output-stream*. It precedes the printed representation with a **Newline** and follows it with a space.

The printed representation of *object* output by **print** includes the escape characters (\ and |) as necessary, in order that they may be read in correctly.

[*Function*]

pprint *object* **&optional** *output-stream* => *object*

This function outputs a printed representation of *object* (to *output-stream*) that is formatted for user readability.

[*Function*]

princ *object* **&optional** *output-stream* => *object*

This function outputs the printed representation of *object* to *output-stream*.

The printed representation of *object* output by **princ** does not include the escape characters (\ and |). This implies that the printed representation may not be read in correctly.

[*Function*]

write-char *character* **&optional** *output-stream* => *char*

This function outputs *character* to *output-stream* and returns *character*.

[*Function*]

terpri **&optional** *output-stream*

This function outputs a **Newline** to *output-stream* and returns **nil**.

[*Function*]

flatsize *object* => *length*

This function returns the number of characters needed for the printed representation of *object* (with necessary escape characters).

[*Function*]

flatc *object* => *length*

This function returns the number of characters needed for the printed representation of *object* (without escape characters).

22.3.2 Output to Binary Streams

[*Function*]

write-byte *integer* **&optional**
    binary-output-stream => *integer*

This function writes *integer*, which represents one byte, to *binary-output-stream*.

integer must be within the range of the type specified by **:element-type** in the call to **open** that created the stream.

Compatibility note: *binary-output-stream* may be either a character stream or a binary stream. The *binary-output-stream* is optional (it defaults to ***standard-output***).

22.3.3 Formatted Output to Character Streams

[*Function*]

format *destination control-string* **&rest**
    arguments => *nil/string*

This function outputs *control-string*, formatted according to both the format directives embedded within *control-string* and the *arguments* following it, to *destination*.

The *destination* must be either **nil**, **t**, or a *stream*. If *destination* is **nil**, **format** creates a string to contain its output and returns that string. If *destination* is **t**, **format's** output is sent to the *stream* that is the value of ***standard-output***, and **format** returns **nil**. Otherwise, if *destination* is a *stream*, **format's** output is sent to it, and **format** returns **nil**.

The *control-string* must be a string. All characters which are not part of a format directive are output as they appear in the *control-string*.

A format directive consists of a tilde character (~), optional colon (:) and atsign (@) modifiers, and a single character (case ignored) specifing the type of directive. Most directives output one or more of the elements in *arguments* formatted according to the directive. The following is a list of supported format directives. In each, the term *arg* refers to the next element in *arguments* to be processed.

~A ascii. *arg* is printed as if by **princ**.

~S s-expression.
arg is printed as if by **prinl**.

~D decimal. *arg* (which must be an integer) is printed in decimal radix with no trailing decimal point.

~B binary. *arg* (which must be an integer) is printed in binary radix.

~O octal. *arg* (which must be an integer) is printed in octal radix.

~X hexidecimal.
arg (which must be an integer) is printed in hexidecimal radix.

~C character.
arg (which must be a character) is printed.

~% newline. A **#Newline** character is printed.

~& freshline.
Identical to **newline**.

~~ tilde. A tilde character is printed.

~<newline> The **Newline** character and any following whitespace is ignored. With a **:**, only the **Newline** is ignored. With a **@**, only the whitespace following the **Newline** is ignored.

~? indirection.
Treats the next *arg* (which must be a string) and the *arg* after it (which must be a list) as a **format** control string and its argument list, respectively.

~[*str0*~;*str1*~;...~;*strn*~] **conditional expression.**
The *arg*th *str* is processed as a format control string. If *arg* is out of range, none of the *strs* are processed; unless the last *str* separator is ~:;, then the last *str* is selected if *arg* is out of range.

~:[*false*~;*true*~] **if-then expression.**
If *arg* is **nil**, *false* is processed as a format control string; otherwise *true* is processed.

~@[*true*~] **test.**
If *arg* is not **nil** then *arg* is not consumed (i.e., it remains the next *arg* to be processed) and *true* is processed as a format control string; otherwise, *arg* is consumed and *true* is ignored.

Compatibility note: Not all directives are supported. *destination* cannot be a string with a fill pointer.

22.4 Querying the User

[*Function*]

y-or-n-p &optional *format-string arguments* => *boolean*

This function is a predicate which is true if and only if the user types a **y** (in upper or lower case) or a **Space** in response to the message specified by *format-string* and *arguments*.

The only other valid responses are **n** and **Rubout**, both of which cause **y-or-n-p** to return **nil**.

[*Function*]

yes-or-no-p **&optional** *format-string arguments* => *boolean*

This function is a predicate which is true if and only if the user types **yes** (in upper or lower case), followed by a **Newline**, in response to the message specified by *format-string* and *arguments*.

The only other valid response is **no**, followed by a **Newline**, in which case **yes-or-no-p** returns **nil**.

Chapter 23

File System Interface

23.1 File Names

23.1.1 Pathnames

23.1.2 Pathname Functions

[*Function*]

pathname *pathname* => *pathname*

This function parses *pathname* and returns an equivalent pathname object.

pathname may be a string, a symbol (whose printname is used), or a pathname object (which is simply returned). No defaulting is done; pathname components which are unspecified in *pathname* are set to **nil** in the pathname object.

Compatibility note: *pathname* cannot be a stream.

[*Function*]

parse-namestring *pathname* => *pathname*

This function parses *pathname* and returns an equivalent pathname object.

pathname may be a string, a symbol (whose printname is used), or a pathname object (which is simply returned). No defaulting is done; pathname components which are unspecified in *pathname* are set to **nil** in the pathname object.

Compatibility note: *pathname* cannot be a stream. No optional or keyword arguments are allowed. Only a single value is returned. Thus, **parse-namestring** is currently identical to **pathname**.

[*Function*]

parse-directory-namestring *name* => *pathname*

This function parses *pathname* as if it named a directory and returns an equivalent pathname object.

pathname may be a string, a symbol (whose printname is used), or a pathname object (which is simply returned). No defaulting is done; pathname components which are unspecified in *pathname* are set to **nil** in the pathname object.

The *name* and *type* components are always set to **nil**.

[*Function*]

merge-pathnames *pathname* **&optional** *defaults* => *pathname*

This function creates and returns a new pathname object that is a copy of *pathname* except that unspecified (i.e., **nil**) components are replaced with components from *defaults*.

pathname and *defaults* may each be a string, a symbol (whose print name is used) or a pathname object; each is converted to a pathname object as if by the function **pathname**. If *defaults* is not provided, it defaults to the value of ***default-pathname-defaults***.

First, all of the specified (i.e., non-**nil**) components in *pathname* are placed in corresponding components in the new pathname object. Then any components which remain unspecified in the new pathname object are filled with the corresponding components in *defaults*.

Compatibility note: The optional *default-version* argument is not supported.

[*Variable*]

default-pathname-defaults => *pathname*

The value of this variable is a pathname object which is the default pathname-defaults pathname.

The value of this variable may be any object acceptable to the function **pathname.**

Any pathname primitive which takes an optional *defaults* argument uses the value of this variable when the *defaults* argument is not provided.

[*Function*]

make-pathname &key :device :directory
:name :type :defaults
=> *pathname*

This function creates and returns a pathname whose components are specified by the keyword arguments.

The component keyword arguments **:device**, **:directory**, **:name**, and **:type** must be either strings, **nil**, **:wild** (for **:name** and **:type** only), or symbols (in which case their print names are used). The given component keyword arguments are placed in corresponding components of the new pathname.

The **:defaults** keyword may be any object acceptable to the function **pathname.** If the **:defaults** keyword argument is provided, those components of the new pathname which were not specified by the component keyword arguments are filled by the components in the **:defaults** keyword argument; otherwise, no defaulting is done.

Implementation note: The directory component may be a list of strings, each string representing a subdirectory of the string to its right.

Compatibility note: The **:host** and **:version** keyword arguments are not supported.

[*Function*]

pathnamep *object* => *boolean*

This function is a predicate which is true if and only if *object* is a pathname object.

[*Function*]

pathname-device *pathname* => *string*

This function returns the device component of pathname.

pathname may be any object acceptable to the pathname function. If *pathname* has a specified device, its name is returned as a string; otherwise **nil** is returned.

[*Function*]

pathname-directory *pathname* => *object*

This function returns the directory component of *pathname*.

pathname may be any object acceptable to the function **pathname**. If *pathname* does not have a specified directory component, **nil** is returned. Otherwise, if the directory component of *pathname* consists of a single subdirectory then a string representing it is returned; otherwise, if the directory component is composed of more than one subdirectory (i.e., it is a hierarchy) then an ordered list of the subdirectories (each represented by a string) is returned.

[*Function*]

pathname-name *pathname* => *object*

This function returns the name component of *pathname*.

pathname may be any object acceptable to the pathname function. **pathname-name** may return either **nil**, **:wild**, or a string, depending upon whether the name component was unspecified, wild, or a specific name, respectively.

[*Function*]

pathname-type *pathname* => *object*

This function returns the type component of *pathname*.

pathname may be any object acceptable to the **pathname** function. **pathname-type** may return either **nil**, **:wild**, or a string, depending upon whether the type component was unspecified, wild, or a specific name, respectively.

[*Function*]

namestring *pathname* => *namestring*

This function returns a string which represents *pathname* in an implementation dependent manner.

pathname may be any object acceptable to the **pathname** function.

[*Function*]

file-namestring *pathname* => *namestring*

This function returns a string which represents the name and type components of *pathname* in an implementation dependent manner.

pathname may be any object acceptable to the **pathname** function.

[*Function*]

directory-namestring *pathname* => *namestring*

This function returns a string which represents the directory component of *pathname* in an implementation dependent manner.

pathname may be any object acceptable to the **pathname** function.

23.2 Opening and Closing Files

[*Function*]

open *pathname* **&key :direction**
:element-type => *stream*

This function returns a new stream that is connected to an external file named by *pathname*.

pathname may be any object acceptable to the **pathname** function. The keyword arguments specify what kind of stream to connect to the file, and how to handle opening the file. A list of keyword arguments and their allowed values follows:

:direction
:input *(default)*, **:output**
:element-type
string-char *(default)*, **unsigned-byte**

An error is signalled if *pathname* is opened in the :input direction and no such file exists.

If *pathname* is opened in the **:output** direction and such a file already exists, it is overwritten.

Compatibility note: Not all values for **:element-type** or **:direction** are currently supported. The **:if-exists** and **:if-does-not-exist** keyword arguments are not currently supported. Version related features are not supported.

[*Macro*]

with-open-file (*Stream pathname* {*option*}*)
{*form*}* **=>** *last-form-result*

This macro establishes a connection between a stream, named by *stream*, and a file, named by *pathname*, within which the forms are evaluated as an implicit **progn**.

stream must be a symbol. The values of *pathname* and each *option* must be acceptable to the **open** function. Each *form* must be a valid form.

The file named by *pathname* is opened as if by open, in compliance with the specified options. The variable named by the symbol *stream* is bound to the resulting stream. Then the forms are evaluated as an implicit **progn** and the value of the last form is returned.

When **with-open-file** is exited, either normally (after evaluation of the last form) or abnormally (e.g., due to a throw), the stream named by *stream* is closed (which also closes the associated file).

Compatibility note: If a new output file is being written to when an abnormal exit occurs, the file is merely closed.

23.3 Renaming, Deleting, and Other File Operations

[*Function*]

rename-file *pathname new-name*
    => *new-name old-truename new-truename*

This function changes the name of *pathname* to *new-name*.

If the file is successfully renamed, three values are returned: the *new-name* pathname with no missing components, the old *truename* of *pathname*, and the new *truename* of *pathname*. Otherwise, if the file cannot be successfully renamed, an error is signalled.

[*Function*]

delete-file *pathname* => *non-nil-result*

This function deletes *pathname* from the file system.

[*Function*]

probe-file *pathname* => *pathname/nil*

This function checks whether or not an external file named *pathname* exists. If one does, the true pathname of the file is returned; otherwise **nil** is returned.

pathname may be any object acceptable to the **pathname** function.

[*Function*]

file-info *pathname* => *attribute filesize-hi filesize-lo creation-date creation-time*

This function returns **PC-DOS** (or **MS-DOS**) encoded information about the file named *pathname*.

23.4 Loading Files

[*Function*]

load *pathname* **&key :verbose :print :if-does-not-exist** => *result*

This function loads the file named by *pathname* into the GCLISP environment.

[*Variable*]

load-verbose => *boolean*

This variable provides the default value for the **:verbose** argument of function **load**.

Implementation note: The initial value is **t**. ***load-verbose*** also affects the behavior of **fasload**.

[*Function*]

fasload *pathname* => *pathname*

This function loads the compiled-code file named *pathname*. If *pathname* has a missing type component, it defaults to **fas**.

If the current value of ***load-verbose*** is non-nil, **fasload** prints the name of the file being loaded in the form of a comment (just like **load**).

[*Macro*]

autoload *Function-name pathname* => *function-name*

This macro causes the file named by *pathname* to be loaded when *function-name* is first used in a function call. The file must contain a definition of *function-name*. After the file is loaded, the evaluation of the function call proceeds normally.

23.5 Accessing Directories

[*Function*]

directory *pathname* => *nil/pathname-list*

This function returns a list of pathnames which match *pathname* (whose components may be wild).

Examples:

(directory "*.*") => *a list of all the pathnames in the current directory*

[*Function*]

cd **&optional** *pathname* **=>** *default-pathname-defaults*

This function changes the **PC-DOS** (or **MS-DOS**) current disk drive and directory to those specified in *pathname*. **cd** also updates the value of ***default-pathname-defaults*** to correspond to the new **PC-DOS** current drive and directory and returns the new value of ***default-pathname-defaults*** as a result.

The argument to **cd** must be either a **pathname** object or a namestring. If *pathname* is a namestring, it is converted to a pathname using **parse-directory-namestring**.

With no arguments, **cd** merely updates the value of ***default-pathname-defaults*** to correspond to the current **PC-DOS** drive and directory and returns the new value of ***default-pathname-defaults*** as a result.

cd is identical to the **PC-DOS** command **cd**, except that the former changes the current drive while the latter does not.

Note that all GCLISP file system functions get the default drive and directory from ***default-pathname-defaults***, *not* from the **PC-DOS** defaults. Thus in GCLISP, there is only one default directory, not one per drive.

Chapter 24

Errors

24.1 General Error-Signalling Functions

[*Function*]

error *format-string* **&rest** *args*

This function signals a fatal (i.e., non-continuable) error.

The error handling system will apply the function **format** to the arguments **nil**, *format-string*, and all the *args*, in order to produce an error message.

[*Function*]

cerror *continue-format-string error-format-string*
&rest *args* => *nil*

This function signals a continuable (i.e., non-fatal) error. If the error is *continued* from (e.g., via the function **continue**), **cerror** returns **nil**.

[*Function*]

break **&optional** *format-string* **&rest** *args* => *nil*

This function suspends the current evaluation state and enters a new Break-Level Loop. If the break is *continued* from (e.g., via the function **continue**), **break** returns **nil**.

[*Variable*]

break-prompt => *function*

The global value of this variable is a function which is called each time through a Break-Level read-eval-print loop.

The function must take no arguments. The values it returns are discarded. The intended purpose of the function is to print a prompt on the ***debug-io*** stream. The function may assume that its output will be printed on a fresh line. The function is called before **read**.

The initial value of ***break-prompt*** is a function which prints the value of ***break-level*** followed by the string "> " on the ***debug-io*** stream. (If the current package is other than *user*, then the value of ***break-level*** is preceded by the name of the current package (followed by ": ")).

[*Variable*]

break-level => *integer*

The value of this variable represents the number of nested break points or errors that are waiting to be handled.

24.2 Specialized Error-Signalling Forms and Macros

24.3 Special Forms for Exhaustive Case Analysis

24.4 Error Handling

[*Special form*]

ignore-errors {*form*}*
    => *nil/last-form-result nil/error-description*

This special form is like a **progn** except that it handles an error by immediately returning **nil** as its first value and a string describing the error as its second value. If no error is signalled while **ignore-errors** is being evaluated, it returns the first result of the last *form* as its first result and **nil** as its second result.

[*Function*]

continue

This function *continues* from an error signalled by **cerror** or a break caused by **break.**

[*Function*]

clean-up-error

This function returns control to the Top-Level or Break-Level Loop that was invoked prior to the most recent error. It ensures that all **unwind-protect** *clean-up* forms are evaluated.

Chapter 25

Miscellaneous Features

25.1 The Compiler

GCLISP does not currently support a compiler.

25.2 Documentation

[*Function*]

documentation *symbol doc-type* => *doc-string*

This function returns the documentation string associated with *symbol* considered as a *doc-type*. If there is no such string, **nil** is returned.

doc-type may be one of the following symbols: **variable**, **function**, or **type**.

Compatibility note: The doc-types **structure** and **setf** are not supported.

[*Function*]

doc *symbol* **&optional** *doc-type* => *nil*

This function prints complete documentation of type *doc-type* for *symbol*.

doc-type may be one of the following symbols: **variable**, **function**, or **type**. If it is omitted, **doc** will attempt to determine which *doc-type symbol* is documented as. If it is documented as more than one *doc-type*, each type of documentation will be printed.

nil is always returned.

[*Function*]

lambda-list *name* **&optional** *dont-search-p*
    => *nil/arglist :not-found*

This function attempts to return information about the argument list (i.,e., lambda-list, parameter list) of the function named by *name*.

name must be a symbol. If *name* does not have a function definition, two values are returned: **nil** and **:not-found.**

If *name* has an interpreted function definition, the actual argument list is returned.

Otherwise, if the function definition of *name* is compiled, the action taken by **lambda-list** depends on the value of *dont-search-p*:

If *dont-search-p* is **nil** (the default), **lambda-list** searches on-line documention file for the function's argument list. If the documented argument list is found, it is returned. Otherwise, two values are returned: **nil** and **:not-found.**

If *dont-search-p* is non-**nil**, the two values **nil** and **:not-found.** are returned.

25.3 Debugging Tools

[*Macro*]

trace {*Function-name*}* => *t/traced-functions-list*

This function causes the evaluation of each function named by a *function-name* to be *traced*.

A *function-name* must be a symbol whose functional definition is a function.

If **trace** is called with no arguments, a list of the currently traced functions is returned. Otherwise, t is returned.

A function may be untraced using **untrace**.

[*Macro*]

untrace {*Function-name*}* => *list*

This function undoes the effect of the **trace** function, i.e., if any of the arguments to **untrace** are currently *traced*, they are *untraced*.

Each of the arguments to **untrace** must be a symbol. If a symbol has a function definition which is *traced*, **untrace** replaces that function definition with the original function definition.

untrace returns a list containing those functions that were actually untraced.

[*Macro*]

step *form* => *form-results*

This macro causes form to be evaluated in a way that allows the user to selectively observe every step of the evaluation. During the evaluation of **form**, the user may type a **?** to get a list of interaction commands.

[*Function*]

backtrace => *nil*

This function displays the contents of the control stack (i.e., the *regular pdl*).

Each form that was given to the evaluator but which has not yet been completely evaluated is displayed on a separate line in reverse chronological order (i.e., the form most recently given to the evaluator is displayed first). Currently, no special forms are displayed.

[*Macro*]

time *form* => *form-results*

This macro times the evaluation of *form*.

form may be any evaluable form. After it is evaluated, the time elapsed during the evaluation is printed on the stream that is the value of ***trace-output***, then the results of *form* are returned.

[*Function*]

describe *object* => *nil*

This function prints useful information about *object*.

object may be any type of object. **describe** prints to the stream which is the value of ***standard-output***. **nil** is always returned.

[*Function*]

room **&optional** *detail-p gc-p* => *nil*

This function prints internal storage management information.

If *detail-p* is non-**nil**, detailed information is printed; otherwise if it is **nil** (the default), only summary information is printed.

If *gc-p* is non-**nil** (the default), the garbage collector is invoked (via the gc function) before any information is gathered; otherwise, if *gc-p* is **nil**, no garbage collection is done.

room prints to the stream which is the value of ***standard-output***. **nil** is always returned.

[*Function*]

ed **&optional** *pathname* => *nil*

This function invokes the GMACS editor.

If *pathname* is **nil** (the default), **ed** simply returns to GMACS, leaving it in the state that existed when it was exited. If this is the first time GMACS has been invoked, a default edit buffer will be created.

Otherwise, *pathname* must be an actual pathname or a namestring (which is converted to a pathname). GMACS will execute the command **find-file** with the pathname as its argument.

[*Function*]

dribble **&optional** *pathname* => *nil*

This function causes all input and output from ***terminal-io*** to be recorded in a file named *pathname*.

When called with no arguments, dribble terminates the recording of input and output and closes the file named *pathname*.

Compatibility note: The streams ***standard-output*** and **standard-input*** are not dribbled.

[*Function*]

apropos *string* **&optional** *package* => *nil*

This function prints a description of each symbol whose print-name contains *string* as a substring.

string may be a string or a symbol (in which case its print-name is used). Note that the empty string ("") is a substring of any string.

If the optional *package* is given, only the symbols accessable in that package are examined. Otherwise, if no package is specified, all packages are examined.

[*Function*]

apropos-list *string* **&optional** *package* => *list*

This function returns a list of symbols whose print-names contain *string* as a substring.

string may be a string or a symbol (in which case its print-name is used). Note that the empty string ("") is a substring of any string.

If the optional *package* is given, only the symbols accessable in that package are examined. Otherwise, if no package is specified, all packages are examined.

[*Variable*]

break-event => *function*

The value of this variable must be function, which will be invoked whenever the user types the *break* key-sequence.

The initial value of this variable is the function name **break**.

25.4 Environment Inquiries

25.4.1 Time Functions

[*Function*]

get-decoded-time => *second minute hour date month year*

This function returns the current time in *Decoded Time* format.

Compatibility note: The values *day-of-week*, *daylight-savings-time-p*, and *time-zone* are not returned.

25.4.2 Other Environment Inquiries

[*Function*]

lisp-implementation-type => *string*

This function returns a string which identifies the generic name of a particular implementation of COMMON LISP.

Implementation note: The string "GOLDEN Common Lisp" is returned.

[*Function*]

lisp-implementation-version => *string*

This function returns a string which identifies the current version of the particular implementation of COMMON LISP.

Implementation note: The implementation version string will have the form,

"*major-version.minor-version description*"

where *major-version* and *minor-version* are both one or two digit numbers and *description* is some text indicating some specialization of the version (e.g., Beta Test). If no *description* is provided, the space following *minor-version* is omitted.

[*Variable*]

features => *list*

The value of this variable is a list of symbols that represent *features* supported by this particular implementation.

Implementation note: The following symbols may appear in the list of features: **gclisp, 8087-fpp.**

25.5 Identity Function

[*Function*]

identity *object* => *object*

This function simply returns the value of *object*. It is used primarily as a functional argument.

25.6 Implementation Specific Procedures and Variables

[*Variable*]

obarray => *array*

The value of this variable is a general array (with a 2 element leader) which is used internally to manage the GCLISP name space. The second leader element contains an association-list which maps macro characters to their respective functions.

25.6.1 Storage Management Functions

[*Function*]

allocate *number-of-paragraphs parts-cons-space parts-atom-space reserve-p* => *integer*

This function allocates additional GCLISP **cons** and **atom** storage space.

number-of-paragraphs must be an integer, which specifies the number of 16-byte paragraphs to allocate or reserve.

parts-cons-space and *parts-atom-space* must both be integers.

Together, they specify that the space to be allocated should be divided according to the ratio *parts-cons-space/parts-atom-space* cons space to atom space. Either integer may be zero (but not both).

If *reserved-p* is **t**, all available memory, except for *number-of-paragraphs* paragraphs, is allocated to GCLISP.

If *reserved-p* is **nil**, *number-of-paragraphs* paragraphs are allocated to GCLISP.

If *reserved-p* is an integer, *number-of-paragraphs* paragraphs are allocated to GCLISP, beginning at address *reserved-p*. This allows the user to specify a memory address that is outside the range of **PC-DOS** (or **MS-DOS**) memory management, e.g., >640K.

Once memory has been allocated to GCLISP, it cannot be returned to the operating system. Note that the cons/atom ratio of allocated memory can only be changed by allocating additional memory with a different ratio.

[*Function*]

gc => *nil*

This function invokes the garbage collector.

If the value of ***gc-event*** is non-**nil**, then it must be a function. The function is called after the garbage collection. Otherwise, if the value of ***gc-event*** is **nil**, the garbage is simply collected. In either case, gc simply returns **nil**.

Also, during a garbage collection, the letters "GC" appear in the lower left hand corner of the display screen if the value of the global variable ***gc-light-p*** is non-**nil**.

[*Variable*]

gc-light-p => *boolean/integer*

The value of this variable is used to control the displaying of the characters "GC" in the lower left hand corner of the display screen during a garbage collection. The value may be one of the following:

nil Do not display the characters.

t Display the characters (white characters on a black background, i.e., using IBM PC character attribute 7).

integer *integer* is used as a bit vector that specifies the IBM PC character attributes to be used in displaying the characters.

The initial value of ***gc-light-p*** is #b01110000 (reverse video).

[*Variable*]

gc-data => *vector*

The value of this variable is an unsigned 8-bit byte vector (with a two-element leader) that contains information about the allocation of memory.

Leader element 0 acts as a gc-in-progress flag. If it contains a non-**nil** object, then a garbage collection is currently in progress. Otherwise, if it contains **nil**, then one is not in progress. Leader element 0 should always contain **nil** when accessed by the user.

Leader element 1 contains an integer that represents the number of garbage collections which have been performed since GCLISP was invoked.

The main vector consists of 9-byte groups, each of which represents a *region descriptor*. Thus, elements 0 through 8 represent region 0, elements 9 through 17 represent region 1, etc.

A region descriptor has the following format:

Byte 0 Region Type. The following type codes are currently supported:

0 - dynamic **cons** space
1 - dynamic **atom** space
2 - static **cons** space
3 - static **atom** space

If the value of this byte is 255, then the previous region descriptor is the last valid descriptor. The value of the bytes following a descriptor byte 0 whose value is 255 is undefined.

Bytes 1-2	Segment Offset Address of Region.
Bytes 3-4	Segment Base Address of Region.
Bytes 5-6	Length of Region.
Bytes 7-8	Number of Free Bytes Remaining After Last Garbage Collection. If this is a dynamic cons region descriptor then these bytes contain the number of free conses remaining.

Note that all 2-byte quantities are stored with the most significant byte at the higher address.

The value of this variable should not be changed in any way.

[*Variable*]

gc-event => *nil/function*

The value of this variable may be either **nil** or a function.

After a garbage collection has been performed (e.g., due to insufficient **cons** space, insufficient **atom** space, or the user's invocation of the gc function), the value of ***gc-event*** is examined.

If the value is a function, it is called with no arguments. Otherwise, the value must be **nil**, in which case nothing is done.

25.6.2 Operating System Interface Functions

[*Function*]

gclisp [*Dos-pathname*] [*/R regular-pdl-size*] [*/S special-pdl-size*] [*/O obarray-size*]

This **PC-DOS** (or **MS-DOS**) command invokes the GCLISP interpreter environment when invoked at **PC-DOS** command level. It is not an actual GCLISP function.

When GCLISP is invoked it creates an initial stack group and an *obarray*. The user can specify the sizes of these objects using the **/R**, **/S**, and **/O** options. Their default sizes are 2000, 500, and 511 32-bit doublewords. It is recommended that *obarray-size* be one less than some power of two.

After these objects are created, the initialization file, named **init.lsp**, is loaded (using **load**) from the current **PC-DOS** directory. If the file is not present, an error is signalled.

Once the initialization file is loaded (using LOAD), if the *dos-pathname* argument is present, the file that it names is loaded. *dos-pathname* must be a valid **PC-DOS** pathname (not a GCLISP pathname).

[*Function*]

exit

This function terminates the current GCLISP environment and returns control to whomever invoked GCLISP (e.g., the operating system command processor).

[*Function*]

dos **&optional** *command-line* => *nil/dos-error-code*

This function invokes the **PC-DOS** (or **MS-DOS**) command processor (e.g., **command.com**).

If **dos** is invoked with no argument, the user is placed at the **PC-DOS** top level command processor. The user may return to GCLISP by executing the **PC-DOS** **exit** function.

If **dos** is invoked with a string or symbol (whose print name is used), it is passed to the **PC-DOS** command proceesor as a command line. When **PC-DOS** is finished processing the command, control returns to GCLISP.

If the **PC-DOS** command processor cannot be invoked (e.g., due to insufficient memory), then the **PC-DOS** internal error code is returned by **dos**. Otherwise, **dos** returns **nil**.

Implementation note: In order for the **dos** function to perform correctly, there must be sufficient memory reserved for the operating system (see the function **allocate**) and the command processor (e.g., **command.com**) must be accessible.

[*Function*]

exec *program-pathname command-string* => *unknown*

This function executes the **PC-DOS** (or **MS-DOS**) *executable* program named by *program-pathname*, passing *command-string* as a command line.

If the **PC-DOS** program cannot be invoked (e.g., due to insufficient memory), then the **PC-DOS** internal error code is returned by **exec**. Otherwise, **exec** returns **nil**.

Implementation note: In order for the **exec** function to perform correctly, there must be sufficient memory reserved for the operating system (see the function **allocate**) and the specified program must be accessible.

25.6.3 IBM PC Specific Functions

[*Function*]

select-page *active-page* => *undefined*

This function selects a new active display page (valid only in **IBM-PC BIOS** *alpha mode*). *active-page* must be an integer in the range 0 to 7 (inclusive) for 40X25 modes and must be an integer in the range 0 to 3 (inclusive) for 80X25 modes.

[*Variable*]

display-page => *active-page*

The value of this variable is an integer which represents the active display page. It is set by the function **select-page**, and should not be set any other way.

[*Function*]

8087-fpp **&optional** *keyword* => *boolean*

This function controls GCLISP's use of the Intel **8087** Numeric Processor Extension.

If the argument is the keyword **:use**, GCLISP will assume that the **8087** is present. (It is an error if it is not.)

If the argument is the keyword **:emulate**, GCLISP will assume that the **8087** is not present, and will emulate it in software.

Otherwise, the argument must be the keyword **:automatic**, in which case GCLISP will check for the presence of the **8087**. If the **8087** is present, GCLISP will utilize it, and **8087-fpp** will return **t**; otherwise, GCLISP will emulate it, and **8087-fpp** will return **nil**.

If no argument is given, then **nil** is returned if emulation is being done and ~~**nil**~~ t is returned otherwise.

25.6.4 Low-Level Functions

The following functions do no error checking. The improper use of any of these functions may violate the integrity of the current GCLISP environment.

[*Function*]

%contents *segment-base-address segment-offset-address*
=> *byte word higher-word*

This function returns the values of the *byte* and *word* stored at the logical address specified by the *segment-base-address segment-offset-address*. **%contents** also returns the value of the next highest word.

Thus, **%contents** effectively returns the byte, word, and double-word at the specified address.

[*Function*]

%contents-store *segment-base-address*
segment-offset-address value data-size
=> *nil*

This function stores *value* at the logical address specified by the *segment-base-address segment-offset-address*.

If *data-size* is **nil**, then *value* is stored in the addressed byte. If *data-size* is **t**, then *value* is stored in the addressed word. Otherwise, *data-size* must be an integer, and *value* and *data-size* are stored in the addressed double-word (*value* is stored in the lower-addressed word).

[*Function*]

%ioport *io-address value word-p* => *in-value/out-value*

This function either transfers *value* to the output port at *io-address* (and returns *value*), or returns the current value of the input port at *io-address*.

If *value* is **nil**, **%ioport** returns the current value of the input port at *io-address*. Otherwise, *value* must be an integer, which is transferred to the output port at *io-address* and returned by **%ioport**.

If *word-p* is **t**, a word is actually being transferred to/from *io-address*+1:*io-address*. Otherwise, if *word-p* is **nil**, a byte is being transferred to/from *io-address*.

[*Function*]

%pointer *object* => *segment-offset-address segment-base-address*

This function returns the logical address of *object*.

[*Function*]

%structure-size *object* => *integer*

This function returns the physical size (in 8-bit bytes) of *object*.

If *object* is of type **fixnum**, 0 is returned since fixnums are represented directly as a special kind of pointer.

[*Function*]

%sysint *interrupt-type ax bx cx dx*
    &optional *ds es*
    => *flags ax bx cx dx*

This function generates a software (i.e., internal) interrupt whose type code is *interrupt-type*. Basically, it executes the **Intel 8086/8088 INT** intruction with *interrupt-type* as its operand.

Before generating the interrupt, **%sysint** loads the **AX**, **BX**, **CX**, **DX**, and optionally the **DS** and **ES** registers with *ax*, *bx*, *cx*, *dx*, *ds*, and *es*, respectively.

Following the return from the interrupt, **%sysint** returns the contents of the **FLAGS**, **AX**, **BX**, **CX**, and **DX** registers.

[*Function*]

%unpointer *segment-base-address*
    segment-offset-address **=>** *object*

This function returns the *object* at the logical address specified by *segment-base-address* and *segment-offset-address*.

There must be a valid object, which has not been garbage collected, at the specified logical address.

Index

GOLDEN COMMON LISP
APPENDICES

Version 1.01

Table of Contents

Appendix A

Error Messages

This is a list of the error messages you can receive from GCLISP, along with a short description of the cause of each message.

DOS Error: <message>
This occurs for certain peripheral I/O DOS commands. "Drive not ready" is a typical <message>.

Unknown array type.
You have attempted to construct an array of a type which is not supported by GCLISP.

<function>: Array reference out of bounds.
Indicates that an array index is beyond the valid bounds of a given array. <function> refers to the function which was called with the improper reference.

Bad array dimension.
You have attempted to construct a multiply-dimensioned array. GCLISP supports only singly-dimensioned arrays.

Bad arg to STRING: <object>
Indicates that the argument <object> cannot be converted to a string. You must use the **coerce** function with the argument.

<function>: Arg not array or named structure: <object>
Indicates that the <function> requires an array or named structure as its argument.

<function>: Array has no leader: <object>
Indicates that a reference to the array leader of <object> has been made, when no leader has been defined.

No place for named structure symbol in array.
The **named-structure-symbol** option was used with **make-array**, and the element type is not T (general) and the array leader size is less than 2.

<function>: bad keyword: <object>
You have supplied <function> with an unrecognized keyword.

SUBSEQ: Inconsistent indices, START: <object>, END: <object>
Usually caused by a **:START** index greater than an **:END** index.

Floating point overflow or underflow.
Floating point overflow or underflow was detected.

Fixnum overflow or underflow.
Fixnum overflow or underflow was detected.

<function>: Wrong number of arguments.
The wrong number of arguments was supplied to <function>.

<function>: wrong type argument: <object>. A <object-type> was expected.
Indicates that <function> requires an argument of type <object-type> to operate correctly.

Special stack overflow.
Indicates that the special stack has overflowed during a computation. You are returned to Top-Level. You may extend the special stack space by allocating a new stack-group with the required size.

Regular stack overflow.
Indicates that the regular stack has overflowed during a computation. You are returned to Top-Level. You may extend the regular stack space by allocating a new stack-group with the required size.

BREAK, (CONTINUE) to continue.
Informs you that break processing has been entered. You may continue the computation by evaluating (CONTINUE).

CONTINUE not inside a BREAK.
You tried to CONTINUE (Ctrl-P) while at Top-Level.

Can't CONTINUE from this error, use CLEAN-UP-ERROR.
You tried to CONTINUE (Ctrl-P) from a non-continuable error, when you should have used CLEAN-UP-ERROR (Ctrl-G).

Back to: <error message>
Indicates that you have "cleaned-up" to a

previous error level described by <error message>.

CONS space full.
Indicates there is no more available CONS space. You are returned to Top-Level. You must either allocate more space or free some of the CONS area.

OBJECT space full.
Indicates there is no more available ATOM space. You are returned to Top-Level. You must either allocate more space or free some of the ATOM space.

OBJECT space too full.
You have requested that an object be allocated from OBJECT space when there is not enough contiguous free space to contain it. You are returned to Top-Level.

Bad argument to FORMAT.
You have presented FORMAT with an improper first argument. This argument must be an output stream, t, or nil.

Bad format directive: <formatting character>
You have entered an unrecognized ~ formatting character.

FORMAT: Improper nesting in ~[construct: <format-string>
Indicates the ~[construct in <format-string> is nested in an improper manner.

FORMAT: non-integer arg given to ~[construct
You have incorrectly given non-integer arguments to the ~[construct. The arguments must be integer values. INTEGERP may be used to insure that this requirement is met.

Unprintable object, type code <object-type-code> at <segment:offset>
An object with an unspecified type code has been presented to the reader. This typically occurs when operations which return some portion of an object's structure have been called with the improper object type. The integrity of the storage system may have been compromised. This is a severe error and may necessitate re-starting GCLISP.

Attempt to return off bottom of stack: <stack-group>
Indicates a RETURN has been attempted with an empty stack group.

Can't resume <object>, it's not a stack group
<object> is not a stack group.

<stack-group> is not a resumable stack group.
<stack-group> is not resumable (i.e., it is empty).

MAKE-STACK-GROUP: bad argument format.
An improper argument has been given to the function MAKE-STACK-GROUP.

Attempt to create too large a stack group.
A request has been made to allocate an object of type stack-group which requires more memory than is currently available.

MAKE-STACK-GROUP: bad option <option>
<option> is not a supported option for MAKE-STACK-GROUP.

CAR or CDR of non-LIST object: <object>
An attempt has been made to take the CAR or CDR of something other than an object of type CONS.

Bad SETF form: <form>
SETF is not available for <form>.

Can't invert SETF reference: <form>
<form> is not a valid place for SETF.

SETF: Reference is different length than pattern:<form>
The template for SETF is not matched by <form>.

Not enough args for <function>
<function> requires more arguments.

Too many args for <function>
<function> requires fewer arguments.

Bad function <object> in internal function dispatcher.
An invalid object has been called via an internal funcall operation.

Wrong number of args while funcalling stack-group <stack-group>
An improper number of arguments has been passed to the function associated with <stack-group>.

Unbound variable: <symbol>
<symbol> does not have a binding in the current environment.

Illegal object in EVAL, type code: <object-type-code> at <segment:offset>
An object with an unsupported object type code has been encountered. Typically indicates an internal system error, or improper use of the low-level memory-accessor functions. The integrity of the storage system may have been compromised. This is a severe error and may necessitate re-starting GCLISP.

Can't EVAL object: <object>
EVAL has been given an improper argument.

Bad function: <object> while evaluating: <form>.
<object> is not a function.

Undefined function: <symbol> While evaluating: <form>
<symbol> has no function binding.

Bad LAMBDA-list: <object> While evaluating: <form>
An improper LAMBDA form has been input to EVAL.

THROW to non-existent tag: <tag>
THROW as been evaluated without a corresponding CATCH outstanding for <tag>.

Illegal tag <tag> to CATCH
<tag> is not an acceptable object type for CATCH.

COND: Bad clause: <object>
An improperly formed COND form has been evaluated.

RETURN-FROM: too many return values: <value-list>
The block returned to is not expecting the number of return values in <value-list>.

RETURN-FROM: name <symbol> not found.
RETURN-FROM an un-established BLOCK.

GO: tag <symbol> not found.
A GO to an undefined LABEL has been attempted.

FATAL ERROR: Stack overflow during GC.
The control stack has overflowed during a garbage collection. There is no recovery from this error. You are returned to DOS.

Bad option to OPEN: <object>
An unsupported option of <object> has been requested of OPEN.

MAKE-WINDOW-STREAM: bad option: <object>
An unsupported option of <object> has been requested of MAKE-WINDOW-STREAM.

Error opening file: <object>
The request to open file <object> could not be honored. The maximum number of files may be open already; or the file may not have been found in the specified location.

File stream not open.
A request has been made to close a file which is not open.

Disk full. The current disk(ette) contains no more room for data.

RENAME-FILE: file not found: <pathname>
A request has been made to rename a non-existent file.

RENAME-FILE: Cannot rename file: <pathname>
A request has been made to rename a file to the name of another, already-existing file.

Can't delete file: <pathname>
A request has been made to delete a file which is either protected or non-existent.

Dot context error.
A "." has been encountered in the input stream in an illegal context.

Comma not inside backquote.
A comma is illegal except inside backquote, a character string, or vertical bars.

Bad "#\" name: <name>
<name> is an unknown character name.

Bad #+/- feature syntax: <feature>
<feature> must be a symbol in a logical expression (consisting of **and**'s, **or**'s, and **not**'s).

EOF while reading S-exp.
An end-of-file has been encountered while reading an open stream.

Can't find a package named <symbol>.
An unknown package name was encountered.

<symbol> is not an external symbol <package>.
External symbols must be declared to be

external.

Attempt to divide by zero.
There was an attempt to divide by zero.

Close parenthesis read at top level.
A mis-match in parentheses has been encountered.

Division by zero.
A division by zero has occurred.

Unknown stream operation: <object>
An unsupported stream operation has been requested.

End of File on Stream: <closure>
A read past the end-of-file of <closure> has been attempted.

Undefined function: <symbol>
There is no function definition of <symbol>.

Undefined macro: <char>
<char> is not a legal macro dispatching character.

Unbound variable: <symbol>
<symbol> is not bound to anything.

TYPE-OF: Illegal object: <object>
The type of <object> is unknown. The integrity of the storage system may have been compromised. This is a severe error and may necessitate re-starting GCLISP.

Can't COERCE <object> to <object-type>
COERCE does not support the requested conversion of type.

Appendix B

Glossary

Allocate To appropriate a computer resource, such as computer memory or a terminal, for a specific task or operation.

Application (of a function) LISP is an applicative language rather than a statement-oriented language. Applying functions to arguments is the principal mode of executing LISP programs.

Array A data structure that organizes the objects it contains along a coordinate system of N dimensions. The user may define the number of dimensions, their sizes, and the type of elements which the array may contain. An array with no special attributes, such as an array header or a fill pointer, is a *simple* array. An array in which each element may be of any type is a *general* array.

ASCII An acronym for American Standard Code for Information Interchange, a seven-bit code for character data transmission. The ASCII set includes control and graphic characters, as well as ordinary letters, digits, punctuation characters, and special symbols.

Association list (A-list) A list of pairs in which each pair is an association between a *key* and a *datum*. The *car* of a pair is the *key*, and the *cdr* is the *datum*.

Atom An elementary entity in LISP. In the early days of LISP, symbols and numbers were atoms; now, any LISP object except a **cons** is an atom. (See also **list** and **s-expression**.)

Backquote The character "`". This instructs the interpreter to inhibit evaluation until a comma (,) is encountered. Backquote is used in constructing lists.

Binary A number system in base 2. In binary, numbers are represented by strings of 0's and 1's.

Binding An operation on the value of a variable which occurs within a particular programming construct such as a *let*. When the binding occurs, the variable's old value is stored away and the variable takes on a new value. When the programming construct is exited, the variable's old value is re-established.

Break A temporary suspension of program execution, invoked in GCLISP by the keychord **Ctrl-Break** or the function **break.**

Break level A level of the **listener** established when a **break** occurs.

Buffer A temporary data storage area in computer memory. A buffer is commonly used during data input, output, and editing operations. (See also **Edit buffer.**)

Byte A basic size unit of data storage in a computer system. Typically eight bits make up a byte.

Character A data type that includes the representations of printed glyphs such as letters and text-formatting characters.

Cons A LISP data type comprised of two components, called a *car* and a *cdr*. Conses are used primarily to represent **lists.**

Control structure
Program language elements used for organizing data processing within a program. Some control structures govern the flow of processing, such as *catch/throw* and *do*; others control the program's access to variables, such as *let* and *label*. Most LISP control structures are written as either *special forms* or *macros*.

Co-routines Programs which can call one another and resume processing where they left off when control is returned to them. (See also **stack group.**)

Cursor A blinking mark on a terminal screen, indicating the point where a character typed on the keyboard will be displayed.

Data — Information represented in a manner that allows communication, interpretation, or processing (by humans or machines).

Data type — A category of LISP data object. Data types include (among others) numbers, characters, symbols, lists, arrays, structures, and functions. An important feature of LISP is that data objects, not variables, are typed. (See also **type**.)

Debug — To detect, pinpoint, and correct programming errors.

Default — An option or value which applies when none has been specified by the user.

Display — A visual presentation of data.

Dotted list — A list whose last **cons** does not have **nil** as the value for its *cdr*. (See also **dotted pair** and **list**.)

Dotted pair — Another name for a **cons**.

Dynamic extent — See **extent**.

Edit — To create or modify a text. Inserting, deleting, and copying characters, words, or lines are typical editing functions.

Edit buffer — A temporary storage area used by an editor. Typically, files are read into an edit buffer, revised or modified in the buffer, and returned to disk.

Editor — A computer program that processes commands for creating and modifying stored text.

Element — An object contained in a list.

Enter — To submit (a command or function) for processing by the computer. For a LISP function, this means typing the command. A DOS command requires the additional action of pressing the Return (or Enter) key.

Eq and Eql — Operations that test for equality. Two objects are **eq** if they are the same object, or if they are *fixnums* with the same value. Two objects are **eql** if they are the same object, or if they are numbers (integer or floating point) with the same value.

Error level A level of the **listener** established when an error occurs.

Evaluation The operation performed by the LISP function **eval**. It is the process of executing a LISP program.

Extent (of a LISP entity) The time interval (in terms of program execution) during which references to the entity may occur. An entity has *dynamic extent* if references may occur at any time in the interval between establishment of the entity and the termination of the establishing construct. An entity has *indefinite extent* if references may occur as long as the entity continues to exist. (See also **scope**).

File A named physical storage area, with its name stored in a directory. A file stores text or a program.

Filename The name of a particular file. Different file systems (in different computers or operating systems) have different conventions for filenames.

Form A LISP language structure which is presented to the evaluator for interpretation.

Function A LISP object that can be applied to other LISP objects, the function's arguments. A function is a procedure which typically takes objects as input (its *arguments*) and returns objects as output (its *values*).

Function call The process of applying a function to its arguments.

Garbage collection The process of reclaiming, and making usable, all unusable parts of the **workspace**. Space is usable if it is available for allocation to new LISP objects.

GMACS The GCLISP editor.

Hexadecimal A number system in base 16. In hexadecimal, numbers are represented by sequences of the ten digits 0 through 9 and the six letters A through F.

I/O stream See **stream**.

Indefinite extent
See **extent**.

Indefinite scope
See **scope**.

Initialization
The process of loading an operating system or a software package into a computer's memory, for the purpose of running it.

Input editor
A feature of an interactive stream (i.e., a stream which connects with the terminal) that allows the user to edit data typed to the screen. An important feature of the GCLISP input editor is that it responds to a set of **keychords** which invoke special actions to interrupt the normal order of processing.

Interpreter
In a LISP programming system, the program which determines how a given form is to be evaluated.

Iteration
The repetition of an action or procedure. Iteration constitutes a basic control structure in most programming languages. LISP provides several iteration facilities, including **do** and **loop**.

Keychord
A combination of keys that executes a command when pressed together. In written descriptions in this document, a keychord is usually represented by hyphenating the two keys. For example, **Ctrl-X** represents depressing the **X** key while the **Ctrl** key is held down.

Key sequence
A keychord followed by a key, or by another keychord.

Lambda-expression
A procedure or function: that is, a list that represents a functional object. The first element of a lambda-expression is the symbol **lambda**; the second element is the *lambda-list*; and the rest of the elements form the *body* of the lambda-expression.

Lambda-list
In its simplest form, a list of variables. More complex lambda-lists involve special keywords (which start with the character "&").

Lexical variable
See **variable.**

List Either an empty list (represented by the symbol **nil**) or a **cons** whose *cdr* component is a list. A list is therefore either **nil** or a chain of conses linked by their *cdr* component and terminated by **nil**. (See also **atom** and **s-expression.**)

Listener The interactive program in the LISP interpreter which implements the *read-eval-print* loop. It reads typed input, assembles LISP objects from the input, evaluates the objects, and prints the evaluation results to the screen.

Loading In LISP, the process of reading and evaluating files. When a file is loaded, each form it contains is evaluated.

Macro A LISP function which serves as a template for translating a LISP form. When a macro is called, a new form is substituted for it and then evaluated in place of the macro call.

Macro character
A character with an associated function. When the LISP reader encounters a macro character, the reader calls the associated function and uses the result of the function in place of the character. (Note that a macro character is unrelated to a **macro.**)

Mark An indicator in the GMACS **edit buffer.** Marks may be used to jump quickly to different points in the buffer and to delimit specific chunks of the buffer for deletion, copying, etc.

Memory The physical part of the computer which may be accessed by programs for storage and retrieval of data.

Memory address
In LISP, a 4-byte value of the form "Segment:Offset", identifying a specific byte location in memory. The **%pointer** function will return the memory address of any LISP object.

Mini-buffer The bottom two lines of the GMACS screen display. Prompts and messages are displayed here.

Mode A means of representing data and processing it (e.g., "binary mode"). Also, a type of environment (e.g., "input mode" or "edit mode").

Multiple values With respect to a LISP form, the characteristic of returning more than one object from a function call.

Nil A constant symbol whose value is always **nil.** It serves as the logical value FALSE. Nil is also used to represent the empty list.

Non-local exit A facility for exiting from a complex process (e.g., a series of nested function calls), using the *catch* and *throw* forms.

Number Collective name for the **data types** which may represent mathematical values: *integer*, *floating-point*, *ratio*, and *complex number*.

Object (LISP object) Any LISP entity that belongs to one or more types of data structure.

Octal A number system in base 8. In octal, numbers are represented by strings of the digits 0 through 7.

Package A COMMON LISP mechanism which provides management of name spaces.

Pathname The full identification of a file in an operating system with a hierarchical file-storage system. The pathname constitutes the complete information needed by the operating system to locate and access the file.

Point A location between adjacent characters in the GMACS **edit buffer** (the position between the current cursor position and the character preceding the cursor). Deletion and insertion in the buffer are done at the point.

Predicate A type of function that tests for some condition involving its arguments, returning the value **nil** if the condition is false, and some non-nil value, usually **T**, if the condition is true.

Pretty-printing
The style of printing implemented by the LISP **pprint** function, which arranges LISP forms on indented lines to make them easier for humans to read.

Print name A string of characters that identifies a particular LISP symbol in a **package.**

Printed representation
The representation of a LISP object in the form of a printed text.

Prompt The character, or character string, displayed on the terminal screen when an interactive program is ready to receive typed input. It shows where the next input entered will be displayed. (The cursor usually appears just to the right of the prompt character.)

Property list One of the components of a symbol. It is a data list that contains zero or more entries, each of which associates a key (called an *indicator*) with another LISP object (called a *value* or sometimes a *property*).

Reader The LISP input language parser. It reads characters from an input **stream**, constructs LISP objects, and returns them.

Readtable A data structure used by the **reader**, containing syntax specifications for input characters.

Recursion The replication of a form within the form itself. An example of recursion is a function calling itself.

Region In the GMACS editor, the text between the **mark** and the **point**. Also, in the GCLISP workspace, the unit of storage management (each region is either a **cons** or an **atom**).

Return In LISP, the action of passing control back to the function which called the current function.

S-expression Short for *symbolic expression*. Either an **atom**, or a **cons** of two s-expressions. The s-expression is the basic entity in all statements in LISP.

Scope (of a LISP entity)
The spatial or textual region of a program

within which references to the entity may occur. An entity has *lexical scope* if references to it can occur only within program portions textually contained within the language construct which establishes the entity. An entity has *indefinite scope* if references can occur anywhere in any program. (See also **extent.**)

Special form A list whose first element is a symbol (its name) and whose syntax is idiosyncratic. Most special forms are control structures. A special form can be regarded as an extension of the evaluator, since it triggers the evaluation of other forms within the special form during the LISP interpretive process.

Special variable
See **variable.**

Stack group A LISP object that contains the history of a particular LISP computation. Stack groups are useful for implementing control structures such as co-routines. When one co-routine calls another, a stack group stores all of the processing information for the first co-routine while the other one executes.

Stream A LISP object that serves as a source or a sink of data. A stream may interface to an external device for input and output operations. It may be input-only, output-only, or both input and output. There are *character streams* for characters and *byte streams* for integers. Typically a stream connects to a file or a device.

Subprimitive A function which manipulates the GCLISP environment at a very low level. Many subprimitives are used to alter hardware-specific features for a particular type of personal computer. A subprimitive usually has a name that begins with the "%" character.

Symbol A LISP data object used to name a variable, a functional definition, or a LISP object with properties. A symbol has these components: a *print name*, a *value*, a *functional definition*, a *property list*, and a *package*.

Tracing A debugging technique that involves printing to the screen the name of a function, together with its arguments and return values, whenever

stream User-written streams are not part of COMMON LISP.

closure The variables closed over by a **closure** are not shared by any other **closure**, even one defined in the same binding environment.

stack-group Stack groups are not part of COMMON LISP.

3 Scope and Extent

GCLISP does not currently support lexical scoping. Thus, there are no lexical (i.e., static, local) variables. All variables are dynamic (i.e., global).

In order to port a GCLISP program to another COMMON LISP environment, all *free* variables (i.e., variables occurring in a binding environment in which they were not established) should be declared *special* using **proclaim**.

4 Type Specifiers

Currently, the only type specifier which is not a standard type specifier symbol is **(unsigned-byte 8)**. Also, the user cannot define new type-specifier abbreviations.

5 Program Structure

5.3 Top-Level Forms

defun The body of the defined function is not enclosed in a **block** construct.

7 Control Structure

7.1 Constants and Variables

function If the argument is a lambda expression, a *lexical closure* is *not* returned. Rather, **function** merely returns the lambda expression unevaluated. GCLISP does not currently support true COMMON LISP closures. A similar, but restricted, type of closure can be created using the **closure** function.

7.2 Generalized Variables

GCLISP provides a simpler, more efficient facility for defining new generalized variables than that defined by COMMON LISP.

setf If *place* is a **getf** form, **setf** may not return the value of *new-value*. Also, subforms of *place* may be evaluated more than once.

7.7 Blocks and Exits

block The *name* established by **block** has dynamic scope.

7.8 Iteration

prog Tags are dynamically scoped. Therefore one can go to a *tag* in a tagbody from a place within the dynamic extent of the tagbody, and yet not within the lexical scope of that tagbody. This feature should not be relied upon, since it will change in the future.

7.9 Multiple Values

values **values** requires at least one arg., i.e., zero values cannot be returned.

values-list If *list* is the empty list (i.e., **nil**) **values-list** returns a single argument, **nil**.

8 Macros

Currently, the expansion of a macro for the first time will cause the macro-call form to be destructively replaced by its expansion. Thus the macro expansion overhead is incurred only once.

8.1 Macro Definition

defmacro The lambda-list keywords **&key**, **&allow-other-keys**, and **&environment** are not currently supported. Embedded lambda-lists may not contain lambda-list keywords. The macro expansion function does not take an environment as a second argument.

9 Declarations

Since GCLISP currently has no compiler, declarations are not necessary. The DECLARE special form exists only for compatibility with other implementations.

9.1 Declaration Syntax

declare The **special** declaration specifier has no effect on the interpreter. Also, declarations (i.e., **declare** special forms) *are* evaluated by the interpreter, but they have no effect.

12 Numbers

12.5 Irrational and Transcendental Functions

The only functions currently supported are ABS and SIGNUM.

12.7 Logical Operations on Numbers

ash Since integers are of fixed size, an arithmetic shift left can cause the sign to change.

12.9 Random Numbers

Random Numbers are currently not supported.

13 Characters

The type **character** is a subtype of **fixnum**. In other words, characters are represented by fixnums (as they are in ZETALISP). Currently, the font attribute is not supported. The *Control* and *Meta* bits are supported.

14 Sequences

Only a limited number of the generic functions on sequences have been implemented.

14.2 Concatenating, Mapping, and Reducing Sequences

some The *sequence* argument must be a list.

every The *sequence* argument must be a list.

14.3 Modifying Sequences

remove-if The *sequence* argument must be a list.

delete-if The *sequence* argument must be a list.

15 Lists

15.2 Lists

push The value returned by **push** is undefined.

pushnew The value returned by **pushnew** is undefined.

15.4 Substitution of Expressions

subst Keyword arguments are not supported.

sublis Keyword arguments are not supported.

15.5 Using Lists as Sets

member Only the **:test** keyword argument is supported.

adjoin No keyword arguments are supported.

15.6 Association Lists

assoc Only the **:test** keyword argument is supported.

rassoc Only the **:test** keyword argument is supported.

17 Arrays

Only vectors are currently supported. Adjustable arrays, displaced arrays, and bit-vectors are not supported. Also, array leaders are not part of COMMON LISP; they are from ZETALISP.

17.1 Array Creation

make-array Not all keyword arguments are supported. **:initial-contents** must be a list.

18 Strings

18.2 String Comparison

string= The arguments must be strings.

string-equal The arguments must be strings.

string< The arguments must be strings.

string-lessp The arguments must be strings.

18.3 String Construction and Manipulation

string-left-trim
Both arguments must be strings.

string-right-trim
Both arguments must be strings.

20 The Evaluator

20.1 Run-Time Evaluation of Forms

evalhook The function bound to this variable does not take an environment argument.

applyhook The function bound to this variable does not take an environment argument. Also, the function is called when special forms are evaluated.

evalhook **evalhook** does not take an environment argument.

applyhook **applyhook** does not take an environment argument.

22 Input/Output

22.1 Printed Representation of Lisp objects

The standard characters ", (,), ', and ; are not implemented as macro characters.

Only the following # constructs are currently supported: ', (, +, -, ., :, B, D, O, S, X, , and |. Character names which follow the # construct may be prefixed with c-, m-, or c-m-.

Currently, only a single readtable is supported.

set-syntax-from-char
No *readtable* arguments are allowed.

set-macro-character
No optional arguments are allowed. The function associated with a macro character returns a second argument to indicate that the macro character should be ignored.

22.3 Output Functions

write-byte The binary output stream argument is optional.

format Not all directives are supported. The destination argument cannot be a string with a fill pointer.

23 File System Interface

The PC-DOS (or MS-DOS) version of GCLISP does not support the *host* or *version* components.

23.1 File Names

pathname The argument cannot be a stream.

parse-namestring
The first argument cannot be a stream. No optional or keyword arguments are allowed.

Only a single value is returned. Thus, **parse-namestring** is currently identical to **pathname.**

merge-pathnames
The optional default-version argument is not supported.

make-pathname The **:host** and **:version** keyword arguments are not supported.

23.2 Opening and Closing Files

open Not all element types are supported. Version related features are not supported.

with-open-file
If a new output file is being written to when an abnormal exit occurs, the file is merely closed.

24 Errors

Currently, all errors signalled by built-in functions are not continuable (i.e., they are unrecoverable).

25 Miscellaneous Features

25.1 The Compiler

A compiler is not yet supported.

25.2 Documentation

The user cannot add documentation to function definitions, variable definitions, etc. (i.e., doc-strings are ignored).

documentation The doc-types **structure** and **setf** are not supported.

25.4 Environment Inquiries

get-decoded-time
Values day-of-week, daylight-savings-time-p, and time-zone are not returned.

Index

November 19, 1984

GOLDEN COMMON LISP
Release Note GCL0100 - 1

This Release Note summarizes the principal undocumented features and known problems in Version 1.00 of GOLDEN COMMON LISP.

Undocumented Features

1. LISP Explorer

1.1. Explorer "Practice World": The Top-Level interface differs from the normal Top-Level in the following ways:

- Of the special keychords displayed by **Alt-H K**, only the four keychords **Alt-H**, **Ctrl-L**, **Ctrl-Break**, **Esc**, and **Rubout** are in effect.
- **Alt-H** displays a different help menu.
- **Ctrl-Break** exits the LISP Explorer.
- An error does not cause a new listener to be invoked.
- The GCLISP command **(exit)** exits from the Practice World back to the LISP Explorer slides.

2. Miscellaneous

2.1. Packages are not fully implemented. The following are currently supported:

- The built-in packages **lisp**, **user**, **keyword**, and **system**.
- The global variable ***package***
- The package functions **find-package**, **intern**, and **find-symbol**.
- The full symbol-qualifier syntax, i.e. **foo:bar**, **foo::bar**, **:bar**, and **#:bar**.

2.2. Irrational and transcendental functions are not fully implemented. **exp**, **sin**, and others are documented on-line and in the Reference Manual, sections 12.5.1 - 12.5.2. Of these, only **abs** and **signum** are currently supported.

2.3. Default directory: Is the same regardless of the drive. Thus if the current default is **A:\dir1\file.ext** and you specify a pathname like **B:foo.bar**, the actual pathname used will be **B:\dir1\foo.bar** even though **dir1** may not exist on drive B:. This feature affects GMACS, LOAD, and any other function that uses pathname defaults.

2.4. The graphic primitive functions **%draw-line** and **%fill** perform as described here; they are not described in the user documentation.

(**%draw-line** *x1 y1 x2 y2 pen func*)

%draw-line forms a line between the display screen coordinates *(x1,y1)* and *(x2,y2)*. X is the horizontal axis, with values increasing from left to right. The X-range is 0 to 319 (for low resolution) or 0 to 639 (for high resolution). Y is the vertical axis, with values increasing from top to bottom of the screen. The Y-range is 0 to 199 for both high and low resolution. Note that all coordinates are absolute, and must reflect the physical coordinate space of the graphics screen. Thus, for the IBM-PC graphics controller, the upper-left corner of the screen is position (0,0) and the lower-right is position (319,199) or (639,199), for low and high resolution respectively.

pen is an integer value from 0 to 3 inclusive, designating the color of the line. *pen* 0 draws the background color; this has limited use, since the drawn line will be indistinguishable from the background itself. The values 1, 2, and 3 correspond to the three colors of the current palette. There are two palettes, each with three colors. These represent the color palettes supported by the IBM-PC graphics controller, one for the background colors and one for the drawing palette.

The *func* parameter overrides the *pen* parameter in selecting the line color. Its allowable values are 0, 1, and 2:

2: use the background color
1: use the exclusive-OR of the current screen color
0: use the color specified by *pen*.

On a monochrome monitor, use the values 3 for *pen* and 0 for *func*.

(%fill *x1 y1 pen func*)

%fill fills the region around screen position *(x1,y1)* with the color of *pen*. The region consists of all points of the screen whose current color is the same as the current color of *(x1,y1)*, and which can be reached from *(x1,y1)* by a path through points whose current color is the same.

Screen addressing is as in **%draw-line.** *pen* and *func* have the same possible values, with the same meanings, as in **%draw-line.**

Certain regions with complicated boundaries may not be filled properly by **%fill.**

Known Problems

1. Installation

   1.1. Neither **check-files** nor **configure-gclisp** handles DOS errors. Therefore, make sure you don't leave a diskette-drive door open or a write-protect tab on a working (backup) diskette.

2. GMACS

   2.1. Redisplay: An edit window may incorrectly display the current edit buffer contents when:

   - the bottom line of the window is a wrapped line and the point is moved to the end of the line; or

   - part of a wrapped line is deleted; or

   - **Ctrl-V** or **Alt-V** is executed, and either the displayed window or the redisplayed window contains a wrapped line; or

   - adding text to the bottom line of the window causes the line to form a continuation line.

   2.2. BEGINNING-OF-DEFINITION (**Ctrl-Z A**):

   2.2.1. When the point is in the first line of a definition, **Ctrl-Z A** repositions the point at the previous definition.

2.2.2. The search for the beginning of definition stops at <u>any</u> '(' in the leftmost column (even within a string).

2.3. INDENT-SEXP (**Ctrl-Z Q**): Does not indent correctly on various forms.

2.4. S-Expression Movement: Multi-line strings are not always handled correctly.

3. <u>Miscellaneous</u>

3.1. **dribble**: Dribbles every character typed, whether or not it was subsequently deleted.

3.2. GCLISP.EXE: Does not take any arguments (contrary to the <u>Reference Manual</u>).

3.3. **macro**: Does no type checking on its first argument. Giving **macro** anything but an unquoted symbol as its first argument can cause a fatal error. (**macro** is used by **autoload** and **defmacro**).

3.4. **allocate**: Allocating less than 18 paragraphs causes a fatal error.

March 15, 1985

GOLDEN COMMON LISP
Release Note GCL0101 - 1

This Release Note accompanies the release of Version 1.01 of GOLDEN COMMON LISP.

Undocumented features and known problems of Version 1.01 include those described in Release Note GCL0100 - 1 (dated November 19, 1984), which accompanied the release of Version 1.00. That note should be reviewed by any user of Version 1.01.

Other undocumented features are as follows:

1. **%draw-line** and **%fill:** These graphics functions (described in Release Note GCL0100 - 1) are undefined until the files **dline.fas** and **fill.fas** have been loaded. These files can be loaded by loading the GCLISP demonstration file **demo.lsp**. They can also be explicitly "fasloaded", by these commands:

   ```
   (fasload "example\\dline")
   (fasload "example\\fill")
   ```

   The files **demo.lsp**, **dline.fas**, and **fill.fas** are in the **\example** directory on the GCLISP Master diskette. In a hard-disk installation, they are in the directory **C:\gclisp\example.**

2. The macros **with-output-to-string** and **with-open-stream** are undocumented. See the COMMON LISP Reference Manual for their specification.

3. **rem** and **mod:** The operator \\ (double-backslash) is undocumented. It implements the COMMON LISP function **rem**, except that the second argument must be an integer and the result is always an integer. The function **mod** has the same behavior as \\; it is undefined until GMACS has been loaded.

4. **allocate**: if *reserve-p* is an integer, it represents the starting address in *paragraphs* (that is, 16-byte units). (See the GCLISP Reference Manual, pages 215 - 216.)

These two corrections apply to the GCLISP Reference Manual:

1. Page 183, last line: for "Appendix F", read "Appendix C, 'The Window System'".

2. Page 221, section 25.6.3, description of the **8087-fpp** function, last line: for "**nil**", read "**t**".