THE HUGO BOOK

HUGO: AN INTERACTIVE FICTION DESIGN SYSTEM

BY KENT TESSMAN

First Edition

THE HUGO BOOK

HUGO: AN INTERACTIVE FICTION DESIGN SYSTEM

Copyright © 2004 by Kent Tessman
The General Coffee Company Film Productions
www.generalcoffee.com

All rights reserved. No part of this book may be used or reproduced in any form or by any means, or stored in a database or retrieval system, without prior written permission of the publisher except in the case of brief quotations embodied in critical articles and reviews.

Warning and Disclaimer

This book is sold as is, without warranty of any kind, either express or implied. While every precaution has been taken in the preparation of this book, neither the author nor the publisher assumes any responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information or instructions contained herein. It is further stated that neither the author nor the publisher is responsible for any damage or loss to any data or equipment that results directly or indirectly from the use of this book.

First Edition

ISBN 0-9735652-0-9

TABLE OF CONTENTS

BOOK 1 THE HUGO PROGRAMMING MANUAL

I. INTRODUCTION 2
- I.a. Why You're Here (or, Just What Is Hugo?) 2
- I.b. Legal Information 2
- I.c. Names And Acknowledgments 3
- I.d. Manual Conventions 4
- I.e. Packing List 4
- I.f. The Truth About Programming 7
- I.g. Working With Hugo 8
- I.h. Getting Started 9
- I.i. Compiler Switches 10
- I.j. Limit Settings 12
- I.k. Directories 13
- I.l. The Hugo Engine 14
- *I.m. What Should I Be Able To Do Now?* 16

II. A FIRST LOOK AT HUGO 17
- II.a. Basic Concepts 17
- II.b. Hello, Sailor! 18
- II.c. Data Types 19
- II.d. Multiple Lines 22
- II.e. Comments 24
- II.f. Compiler Errors And Warnings 25
- II.g. Compiler Directives 27
- *II.h. What Should I Be Able To Do Now?* 31

III. OBJECTS 33
- III.a. Getting To Know Your Objects 33
- III.b. The Object Tree 34
- III.c. Attributes 39
- III.d. Properties 42

III.e. Classes **48**

III.f. What Should I Be Able To Do Now? **51**

IV. HUGO PROGRAMMING 52

IV.a. Variables **52**

IV.b. Constants **54**

IV.c. Printing Text **56**

IV.d. More Formatting Sequences **61**

IV.e. Operators and Assignments **64**

IV.f. Efficient Operators **66**

IV.g. Arrays And Strings **68**

IV.h. Conditional Expressions and Program Flow **73**

IV.i. What Should I Be Able To Do Now? **79**

Example: Mixing Text Styles 79

Example: Managing Strings 79

V. ROUTINES AND EVENTS 82

V.a. Routines **82**

V.b. Property Routines **85**

V.c. Before And After Routines **88**

V.d. Init And Main **93**

V.e. Events **95**

V.f. What Should I Be Able To Do Now? **96**

Example: "Borrowing" Property Routines 96

Example: Building a (More) Complex Object 97

Example: Building a Clock Event 98

VI. FUSES, DAEMONS, AND SCRIPTS 100

VI.a. Introduction **100**

VI.b. Fuses And Daemons **100**

VI.c. Scripts **102**

VI.d. A Note About The `event_flag` Global **104**

VI.e. What Should I Be Able To Do Now? **105**

Example: A Simple Daemon and a Simpler Fuse 105

VII. GRAMMAR AND PARSING 107

VII.a. Grammar Definition **107**

VII.b. The Parser **112**

VII.c. What Should I Be Able To Do Now? 116
VIII. JUNCTION ROUTINES 117
VIII.a. Before We Get To The Routines 117
VIII.b. Parse 118
VIII.c. ParseError 119
VIII.d. EndGame 121
VIII.e. FindObject 121
VIII.f. SpeakTo 122
VIII.g. Perform 124
IX. THE GAME LOOP 126
IX.a. Overview Of The Game Loop 126
IX.b. What Should I Be Able To Do Now? 128
X. USING THE OBJECT LIBRARY 130
X.a. Rooms and Directions 130
X.b. Characters 132
X.c. Character responses 133
X.d. Scenery and Components 135
X.e. Doors 136
X.f. Vehicles 137
X.g. Plural and Identical Objects 139
X.h. Attachables 142
X.i. What Should I Be Able To Do Now? 145
XI. ADVANCED FEATURES 146
XI.a. The Display Object 146
XI.b. Windows 147
XI.c. Reading And Writing Files 148
XI.d. Mouse Input 151
XII. RESOURCES 152
XII.a. Creating And Using Resources 152
XII.b. Pictures 153
XII.c. Sound And Music 154
XII.d. Video 155

APPENDIX A: SUMMARY OF KEYWORDS AND COMMANDS 157

APPENDIX B: THE HUGO LIBRARY 179

ATTRIBUTES 179
GLOBALS 180
ARRAYS 181
CONSTANTS 181
PROPERTIES 183
VERB ROUTINES 188
UTILITY ROUTINES, ETC 189
AUXILIARY MATH ROUTINES: 202
STRING ARRAY ROUTINES: 203
FUSE/DAEMON ROUTINES: 204
CHARACTER SCRIPT ROUTINES: 205
CHARACTER ACTION ROUTINES: 206
CONDITIONAL COMPILATION: 206

APPENDIX C: LIMIT SETTINGS 207

APPENDIX D: HUGOFIX AND THE HUGO DEBUGGER 209

The HugoFix Debugging Library 209
The Hugo Debugger 212

APPENDIX E: PRECOMPILED HEADERS 215

APPENDIX F: HUGO VERSIONS 218

APPENDIX G: ADDITIONAL RESOURCES 219

BOOK 2 TECHNICAL SYSTEM SPECIFICATION

I. INTRODUCTION 223

I.a. How Hugo Works 223

II. ORGANIZATION OF THE .HEX FILE 225

II.a. Memory Map 225

II.b. The Header 226

III. TOKENS AND DATA TYPES 228

III.a. Tokens 228

III.b. Data Types 229

IV. ENGINE PARSING 233

V. GRAMMAR 235

VI. EXECUTABLE CODE 237

VI.a. A Simple Program 237

VI.b. Expressions 238
VII. ENCODING TEXT 241
VIII. THE OBJECT TABLE 242
VIII.a. Objects 242
VIII.b. Attributes 242
IX. THE PROPERTY TABLE 244
IX.a. Before, After, and Other Complex Properties 244
X. THE EVENT TABLE 247
XI. THE DICTIONARY AND SPECIAL WORDS 248
XI.a. Dictionary 248
XI.b. Special Words 248
XII. RESOURCEFILES 249
XIII. THE HUGO COMPILER AND HOW IT WORKS 250
XIII.a. Compile-Time Symbol Data 252
XIII.b. The Linker 255
XIV. THE HUGO ENGINE AND HOW IT WORKS 257
XIV.a. Runtime Symbol Data 259
XIV.b. Non-Portable Functionality 260
XIV.c. Savefile Format 261
XV. DARK SECRETS OF THE HUGO DEBUGGER 263
XV.a. Debugger Expression Evaluation 264
XV.b. The .HDX File Format 264
APPENDIX A: CODE PATTERNS 266

INDEX 293

AUTHOR'S FOREWORD

Somewhere along the way this became a real book, and a real book deserves a foreword, and maybe even a dedication. Looking back, the reason any of this exists at all probably has something to do with being ten years old, and me and my little brother Dean sitting in front of the family Apple II Plus computer, one chair and one stool, playing those old text adventures.

So Dean, this is for you. I'm sorry I always took the chair.

Kent Tessman
Toronto, Canada
2004

BOOK 1

THE HUGO PROGRAMMING MANUAL

OR

HOW TO WRITE GAMES AND INFLUENCE PEOPLE

I. INTRODUCTION

I.a. Why You're Here (or, Just What Is Hugo?)

Chances are if you're reading this book you're already at least a little familiar with adventure games, and maybe even more specifically interactive fiction or text adventures.[1] Hugo is a system for designing, programming, and running these. It is not the first such system—and it's difficult to find substantial fault in any general way with the best of those systems that predate Hugo[2]—but Hugo does hope to extend the concepts developed in earlier, similar systems in order to make interactive fiction programming less cryptic, and more flexible and accessible to designers, as well as to add functionality in certain areas where other systems are lacking.

What does it mean to be a "system" for interactive fiction? In Hugo's case, it means that not only does it provide an environment for running Hugo games—the rather exciting-sounding Hugo Engine—but also the means of creating them (the Hugo Compiler) and a tool for troubleshooting (the Hugo Debugger). Additionally, it includes the Hugo Library, in essence a suite of Hugo programming code providing the basic infrastructure for a Hugo game.

This book will serve as a means of becoming familiar with what Hugo is and what it does, and what is required to develop an interactive fiction game using Hugo, whether or not you have any prior programming experience.

I.b. Legal Information

Please see the Hugo License for detailed legal information. Hugo is copyrighted by its Author. Programs created using the Hugo Compiler are the property of the individual user who created them. The use of the Hugo library files (the "Hugo Library") and the distribution of the Hugo Engine are

[1] If not, or if you'd like some additional interesting reading, there are a number of excellent resources to investigate further, some of which are listed in *APPENDIX G: ADDITIONAL RESOURCES*.

[2] The best and most popular of these earlier systems are TADS (Mike Roberts, 1987) and Inform (Graham Nelson, 1993).

authorized for the creation of non-commercial or shareware-based software. The use of the Hugo Library is allowed in commercial software, although copyright of the library files themselves remains with the Author. Commercial distribution of the Hugo Compiler, the Hugo Engine, and/or the Hugo Debugger may be allowed by arrangement with the Author. The source code for the Hugo Compiler, the Hugo Engine, and the Hugo Debugger (the "Hugo Source Code") is available for porting to new platforms. Public distribution of modified versions of the Hugo Source Code is not permitted.

Note: The Hugo Compiler, the Hugo Engine, the Hugo Debugger, the Hugo Library, and related components are available free of charge; there is no warranty whatsoever pertaining to their use.

I.c. Names And Acknowledgments

Those who have taken upon themselves the task of porting Hugo to various platforms include Julian Arnold (Acorn/RiscOS port), Gerald Bostock (OS/2 port), David Kinder (Amiga port), Bill Lash (original Unix/Linux port), Andrew Plotkin (Macintosh port using his Glk library), and Colin Turnbull (original Acorn Archimedes port). The author is considerably indebted to them, for all their work as well as for their input on how to improve the compiler and engine. Without their efforts, Hugo and the games created with it would not be available for so nearly as wide an audience.[3]

More than a few words of appreciation must be given to Volker Blasius, the original maintainer of the Interactive Fiction Archive at GMD, one of the key resources for interactive fiction players and developers, and a primary hub of material for contributors to (and readers of) the Usenet newsgroups *rec.arts.int-fiction* and *rec.games.int-fiction*. For years, Volker (earlier with the help of David M. Baggett and later with the help of David Kinder) undertook the substantial task of organizing and cataloguing thousands of existing files and a steady stream of new submissions. The IF Archive is now, as of this writing, housed on the web at *http://www.ifarchive.org*, and is currently maintained by David Kinder and Stephen Granade.

Thanks also to those whose comments and suggestions have contributed to making Hugo as powerful and usable as it is: Torbjörn Andersson, Julian Arnold, Dmitry Baranov, Mark Bijster, Jonathan Blask, Cam Bowes, Jason Brown, Daniel Cardenas, Jose Luis Cebrian, Gilles Duchesne, Jason Dyer, Miguel Garza, Jeff Jenness, Doug Jones, Alan MacDonald, Cena Mayo, Jesse McGrew, John Menichelli, Iain Merrick, Jim Newland, Jerome Nichols, Jason C. Penney, Giacomo Pini, Andrew Pontious, Vikram Ravindran, Gunther Schmidl, Robb

[3] Other ports done by the author are for Windows, Linux, Macintosh, DOS, BeOS, Pocket PC, and PalmOS.

Sherwin, Christopher Tate, Mark J. Tilford, Paolo Vece, and Dean Tessman, as well as to many other Hugo users. Graham Nelson's Inform language helped give early shape to some of the ideas in Hugo's development with regard to syntax and structure. Finally, sincere apologies on my part for any omission of those who have contributed to Hugo over the years in any way.

And thank you, as always, to Jennifer.

I.d. Manual Conventions

Please refer to the following conventions as they are used in this manual:

`<parameter>`	for required parameters
`[parameter]`	for optional parameters
`file`	for specific filenames
`FunctionName`	functions, etc.
Note	important notes related to the matter at hand
`Output`	for output by the compiler or engine
`token`	tokens, keywords
`...`	for omissions (particularly of non-relevant sections of code)

I.e. Packing List

A number of files are part of the basic Hugo package. You'll need to make sure to have these before you get started; a good starting point is the Hugo web page at *http://www.generalcoffee.com/hugo*.

Executable package. You'll need, first and foremost, a version of Hugo compiled for your particular computer system, which will allow you to run existing Hugo programs, as well as compile and run your own. Usually the package itself is named something like:

hugov31_win32.zip	(Windows)
hugov31_macos.sit	(Macintosh)
hugov31_unix_source.tar.gz	(Unix sources)
etc.	

although filenaming may vary between platforms. Generally, like in the examples above, Hugo comes in an archive file containing the various executables for a given platform. The package should contain the following files (although, again, filenames may differ; they'll generally appear as **filename**, although on your system they may be lowercase or some combination of upper and lowercase, and the filename extension may vary or be absent):

Hugo Compiler	(**HC.EXE, hcwin, hc**)
Hugo Engine	(**HE.EXE, hewin, he, hewx**)
Hugo Debugger	(**HD.EXE, hdwin, hd**)
Debugger help file	(**HDHELP.HLP**)

Please note that the Hugo Compiler and the Hugo Debugger are not available for all systems; some packages for some systems contain only the Hugo Engine for playing Hugo games. To develop and compile your own games, the Hugo Compiler is necessary. The Hugo Debugger is a useful and powerful tool, but it is not essential for Hugo development.

Library package. You may be relieved to learn that you don't have to write every last part of a Hugo game yourself. In fact, much of the basic infrastructure is provided by the Hugo Library, a set of existing Hugo source code files that you include in your game to manage the game world. Using the Hugo Library, you can easily create a small game that incorporates the basic behavior of a standard Hugo game. Normally these files can be found in a single archive called **hugolib.zip**:

hugolib.h	Library definitions and routines
verblib.h	Standard verb routines
verblib.g	Standard verb grammar definitions
objlib.h	A library of useful object definitions (included by **hugolib.h**)

The library also includes these three less commonly used files:

resource.h	Resource-handling routines
system.h	System-level routines
window.h	Text window management

Additionally, the library contains two sets of files that, depending on user-specified settings, are optionally included by **hugolib.h:**

hugofix.h	Debugging routines
hugofix.g	Debugging grammar
verbstub.h	Additional verb routines
verbstub.g	Additional verb grammar

Sources. It's probably a good idea as you delve into Hugo programming to have some existing source code to look at. **sample.hug** is a valuable resource to have handy since it contains examples of most aspects of Hugo programming. Additionally, you're probably want to download **shell.hug**, which provides the very bare bones of a Hugo game for you to start building on:

sample.hug	Sample game source code
shell.hug	Empty source code to build on

An additional Hugo source file demonstrates the ability to create precompiled headers (and not something you probably need to worry about just now; it's covered in *APPENDIX E: PRECOMPILED HEADERS*):

hugolib.hug	To create a linkable version of **hugolib.h**

Extras. The last essential remaining piece you'll need to begin Hugo development in earnest is a *text editor* of some sort. This is what you'll use to edit the Hugo source files that you'll write and ultimately compile into working Hugo programs. On Windows or Macintosh you could use the pre-packaged Notepad or SimpleText (or TextEdit on Mac OS X) applications, respectively, but it's really not recommended: there are far better inexpensive or even freeware editors available (and once you get

deeper into programming, you'll realize that the one sure investment you can make is an editor you're comfortable with). On Unix-ish systems (including Linux), you'll generally have a choice of editors including Emacs, vi, and a number of graphical user interface (GUI) programs. It's a little beyond the scope of this book to even attempt to recommend an editor—since it's as much a matter of personal preference as anything—so the best advice that can be given is to ask around, experiment, and find out what works best for you.

It would also be good preparation to become familiar with the *terminal* or *console* on your system. On Windows, this is the "MS-DOS Prompt" or "Command Prompt" under the Start menu, or type "command" (Windows 95/98) or "cmd" (Windows NT/2000/XP) from the "Run..." option; on Unix systems, this will be bash or tcsh or some other kind of command shell. Other systems will have different names for their command-line environments (although on something like a pre-OS X Macintosh, there is no such thing as a terminal or console, so you needn't worry about it).

I.f. The Truth About Programming

The truth about writing interactive fiction games is that yes, it is programming, and no, there's really no way around it. It's impossible for a game design system to provide a cookie-cutter means of picking and choosing all the various facets of any relatively complex game so that by clicking on a few buttons a fully formed and entirely original game world and story will be produced. It doesn't work that way. The attempt to determine at the outset all of the various game elements that will ever be needed by any game author in any type of game necessarily limits what authors are able to include in their games, as well as their ability to tailor gameplay, presentation, character interaction, geography, and other important aspects of a game to the needs of the particular work of interactive fiction they're writing. So, in order to write the best interactive fiction games you're capable of, you'll need to do a at least a little programming. But that's not reason to fret.

The word "programming" seems to hold a sort of mystique that, to the non-programmer, conjures up some unfathomable combination of knowledge and skills that shall remain forever inaccessible to outsiders. In fact, that's pretty far from the truth. Programming is indeed a creative pursuit, but it is pretty much unique among creative pursuits in that it's the only one that can be overcome by enough banging of keys: eventually you can make almost anything work.

If you've never done any programming before, you can probably expect to be slightly baffled by at least some of the early going in this manual. The truth about *learning* programming is that you're probably not going to be able to read through this book (or any book on programming in any other programming language, for that matter) once, in proper sequence, from cover to cover, and be able to write programs expertly in the language. Many of things will require the introduction of concepts that will only be discussed in full later on once a better grounding in the language is achieved. There will, in fact, be several places in this book (especially in the early sections) where readers will be encouraged to not worry if the subject matter at hand seems quite foreign. But rest assured that, after a brief initial period of acclimation, before long things like "objects", "properties", "routines", "global variables", "calling parameters", and a host of others will be rolling off your tongue like the alphabet.

To make everything even easier, Hugo is designed so that writing very basic games will consist largely of defining and describing objects and locations in a very straightforward manner. All of the complex inner workings of the game—from the templates for standard rooms and objects and their related behaviors; to what happens when a player types `>GO NORTH` or `>OPEN THE CARDBOARD BOX` or any other command, recognized or unrecognized; to the rules of the game world for containment, edibility, bulk, switching things on or off, or any number of "physical" traits—are handled by the Hugo Library, and a prospective doesn't have to worry about where these things are handled or how until he or she is ready to investigate deeper.

I.g. Working With Hugo

The way Hugo works is fairly standard for a modern programming language. A programmer begins with a *source file*, which is a human-readable text file (created and edited in a separate text-editing application). The source file contains all the various definitions, instructions, and other text that will ultimately form the content of the game. The content of a source file is formatted in the particular structure of the *Hugo language*—the programming language with which the majority of this manual will endeavor to help you become acquainted.

The programmer inputs the source file to a *compiler* (here, specifically, the Hugo Compiler), which takes the source code and generates an *object file*. The object file is—unlike a source file—not human readable, but has instead been translated by the compiler into a series of optimized instructions that are easily understood by the computer. The computer can then take that object file and execute it as a program, just like any application users regularly use (applications—like word processors and spreadsheets and browsers—which were probably produced by a compiler in exactly the same process as described

here). The difference between a Hugo-generated program and such other compiled programs is that a Hugo program may, once compiled, be run on any platform for which the Hugo Engine exists. Normally a compiled program can only be executed on the platform for which it was compiled; Hugo programs are much more portable, and can be compiled on one platform and subsequently be run on any other of the large number of platforms that Hugo supports.

The Hugo Engine is the *interpreter* or *runtime* for compiled Hugo object files (also referred to as *.HEX files*, after their default extension meaning "Hugo executable"). It functions as a hosting environment in which to load the .HEX file, in sort of the same way that a browser loads a web page from the Internet.

I.h. Getting Started

Let's take the first step by becoming acquainted with the tools we'll be using. First and foremost is the Hugo Compiler. Compiler usage instructions may vary slightly depending on what computer and operating system you're using.

If you're using a GUI version of the compiler (such as the one for Windows), when you start the compiler it will display a form for you to enter the name of the Hugo program you want to compile, along with any other compilation options.

If you're running a command-line version of the compiler, it will behave pretty much the same regardless of what system you're on. Type

```
hc
```

without any parameters to get a full listing of available compiler options and specifications. For example, the Unix and MS-DOS syntax for running the compiler is

```
hc [-switches] <sourcefile[.hug]> <objectfile>
```

It is not absolutely necessary to specify any switches, the name of the objectfile, or the sourcefile extension. The bare-bones version of the compiler invocation is

```
hc <sourcefile>
```

With no other parameters explicitly described, the compiler assumes an extension of **`.hug`**. The default object filename is **`<sourcefile>.hex`**.

Here's how to compile the sample game from the **`sample.hug`** source code mentioned earlier in *I.e. Packing List*. Make sure the compiler executable,

library files, and sample game source code are all in the current directory, then type

```
hc -ls sample.hug
```

or simply

```
hc -ls sample
```

and after a few seconds (or more, or less, depending on your processor and configuration) a screenful of statistical information will appear following the completed compilation (because of the -s switch). The new file **`sample.hex`** will have appeared in current directory. As well, the -l switch wrote all compile-time output (which would have included errors, had there been any) to the file **`sample.lst`**.

Note: The next three sections—*I.i. Compiler Switches*, *I.j. Limit Settings*, and *I.k. Directories*—may seem a little confusing to those without much compiler experience. Do look them over, but if you're not exactly sure what it all means, don't worry about it. You won't need to tell the compiler to do anything particularly acrobatic at the outset, and the information is here for experimentation and for when you need it.

I.i. Compiler Switches

A number of *switches* may be selected via the invocation line. These are one or more single-letter (usually, at least) options that follow a – character. The available options are:

-a	**A**bort compilation on any error
-d	compile as an .HDX **D**ebuggable executable
-e	**E**xpanded error format
-f	**F**ull object summaries
-h	compile in .HLB precompiled **H**eader format
-i	display debugging **I**nformation
-l	print **L**isting to disk as **`<sourcefile>.lst`**
-o	display **O**bject tree
-p	send output to standard **P**rinter
-s	print compilation **S**tatistics
-t	**T**ext to listfile for spellchecking
-u	show memory **U**sage for objectfile
-v	**V**erbose compilation

-w **W**rite **<objectfile>** despite any errors
-x ignore switches in source code
-25 compile v**2.5** with compatibility

- The **-a** switch to abort compilation on any error is useful particularly when you suspect that an error earlier in the program is triggering a string of compilation errors later on. Using -a will stop compilation after the first error.
- In order to compile a file usable with the Hugo Debugger (which means it will contain a large amount of symbolic information not normally included in a .HEX file), use the **-d** switch.
- The standard format in which the Hugo Compiler reports errors is relatively concise, but can sometimes be used by more advanced editors to automatically locate the error-causing line. To have the compiler print errors in greater detail than this standard format, use the **-e** switch.
- Using the **-f** switch will tell the compiler to output a list of detailed information about each object, which can sometimes be useful for debugging.
- The **-h** switch is used to generate a precompiled header, described in *APPENDIX E: PRECOMPILED HEADERS*.
- The **-i** switch tells the compiler to finish compilation by printing a list of all symbols used, as well as their numerical equivalents and any address information. Again, this can sometimes be useful in debugging.
- Most programmers will probably make use of the **-l** switch to record all compilation output to a listfile, by default called **<filename>.lst.** Such recorded output will contain not only any compile-time errors, but also any output generated by the use of other switches listed here.
- To get a list of all objects (as well as a visual depiction of their inheritance), use the **-o** switch.
- The **-p** switch does not exist in all versions of the Hugo Compiler for all platforms. Where present, it causes all output to be sent to a named printer, such as "LPT1" under DOS or Windows, or "/dev/lp" under Unix. (The -p switch is actually deprecated, as it's much easier and more flexible to capture output to a listfile using the -l switch, then subsequently view and/or print the listfile using a text editor program.)
- Compilation statistics are printed as a summary when compilation is done if the **-s** switch is used. The summary includes totals of lines compiled, the numbers of objects, routines, properties, dictionary words, and other elements of a .HEX file.
- The **-t** switch sends all textual output and dictionary entries to the listfile so that it can be run through a spellchecker.

- The **-u** switch gives a breakdown of the memory used by the .HEX file for various things including the object table, the property table, and executable code.
- When the **-v** switch (not available on all versions) is used, the compiler runs in verbose mode and maintains a real-time display of the number of lines compiled, and of the percentage of compilation complete.
- Normally if the compiler encounters any errors in the source code, it won't generate the gamefile. Use the **-w** switch to generate **`<objectfile>`** regardless of any errors encountered. This is useful in a situation where you want to try out a section of code that has nothing to do with another section that may currently have errors, but is otherwise rarely used (for obvious reasons—it's always best to get rid of those pesky errors).
- The version 3.0 (or later) compiler may be invoked with the **-25** switch in order to generate a v2.5 gamefile. Note, however that it's generally unnecessary to do so, since v2.5 and v3.x are compatible; i.e., the v3.0 (or later) engine will run v2.5 gamefiles, and most recent v2.5 builds of the engine will run v3.0 gamefiles. See *APPENDIX F: HUGO VERSIONS* for more information.

I.j. Limit Settings

Also included on the invocation line before the sourcefile may be one or more limit settings. These settings are primarily for memory management, and limit the number of certain types of program elements, such as objects and dictionary entries. In order to allow the compiler to function optimally across a range of different computer platforms with differing memory management capabilities, the compiler does not automatically allow an unlimited number of all language elements. For the most part, you won't need to worry about upping any of these settings until your Hugo games begin to reach larger sizes.

To list the settings, type:

```
hc $list
```

You'll see something like:

```
----------------------------------------------------------
Static limits (non-modifiable):
        MAXATTRIBUTES     128   MAXGLOBALS          240
        MAXLOCALS          16
----------------------------------------------------------
Default limits:
        MAXALIASES        256   MAXARRAYS           256
        MAXCONSTANTS      256   MAXDICT            1024
```

```
        MAXDICTEXTEND     (0)   MAXDIRECTORIES     16
        MAXEVENTS         256   MAXFLAGS          256
        MAXLABELS         256   MAXOBJECTS       1024
        MAXPROPERTIES     254   MAXROUTINES       320
        MAXSPECIALWORDS    64

Modify non-static default limits using:  $<setting>=<new
limit>
----------------------------------------------------------
```

To change a non-static limit (and compile a source file), type:

```
hc $<setting>=<new limit> <sourcefile>...
```

Note: Users of Unix or similar systems (including OS X, BeOS, and others) may, depending on the shell being used, need to escape special tokens like '$' or enclose these arguments in single quotes (e.g. `\$list` and \$<setting>=<new limit> or 'list', '$<setting>=<new limit>', etc.) to override the shell's parsing of those tokens in the compiler invocation line. (Non-Unix users probably don't need to worry about what that means.)

For example, to compile the sample game with the maximum number of dictionary entries doubled from the default limit of 1024, and with the -l and -s switches set,

```
hc -ls $MAXDICT=2048 sample
```

If a compile-time error is generated indicating that too many symbols of a particular type have been declared, it is probably possible to overcome this simply by recompiling with a higher limit for that setting specified in the invocation line.

See *APPENDIX C: LIMIT SETTINGS* for a complete listing of valid limit settings.

I.k. Directories

It is possible to specify where the Hugo Compiler will look for different types of files. This can be done in the command line via:

```
hc @<directory>=<real directory>
```

For example, to specify that the source files are to be taken from the directory **`c:\hugo\source`**, invoke the compiler with

```
hc @source=c:\hugo\source <filename>
```

Valid directories (which can be listed using "**`hc @list`**") are:

`source`	Source files
`object`	Where the new .HEX file will be created
`lib`	Library files
`list`	**`.lst`** files
`resource`	Resources for a `resource` block
`temp`	Temporary compilation files (if any)

Note: Again, users of Unix or similar systems may, depending on the shell being used, need to escape special tokens like '@' or enclose these arguments in single quotes (e.g. `\@list` and `\@<directory>=<real directory>` or `'@list'` and `'@<directory>=<real directory>'`) to override the shell's parsing of those special tokens in the compiler invocation line.

Advanced users may take advantage of the ability to set default directories using environment variables. (The method for setting an environment variable may vary from operating system to operating system.) The **`HUGO_<NAME>`** environment variable may be set to the **`<name>`** directory. For example, the source directory may be set with the **`HUGO_SOURCE`** environment variable. Command-line-specified directories take precedence over those set in environment variables. In either case, if the file is not found in the specified directory, the current directory is searched. (And if you're not familiar with environment variables, again, don't worry about it.)

I.l. The Hugo Engine

Once the sample game has been successfully compiled, you can run it with the help of the Hugo Engine. The way in which you do this will vary depending on what platform you're using.

1. If you're running a GUI version of the engine (such as for Windows), the filetype for .HEX files will generally be associated with the Hugo Engine application, so that double-clicking on the compiled .HEX file will automatically start the engine.

2. Most GUI versions also have the functionality that, if you start the Hugo Engine application directly with no .HEX file given, it will present you with a file-selector to choose the file to run.

3. Command-line versions of the engine require you to specify the name of the .HEX file you want to run. Having compiled the sample game, run it by invoking

```
he sample
```

at the command line (replacing "**he**" with the name of the engine executable for your system, if necessary). Again, it should not be necessary to specify the extension. The engine assumes **.hex** if none is given.

Note: If you know how to set environment variables for your system, the environment variable HUGO_OBJECT or HUGO_GAMES may hold the directory that the Hugo Engine searches for the specified .HEX file. The location for save files may be specified with HUGO_SAVE. All of these are optional.

I.m. What Should I Be Able To Do Now?

By now, you should be able to:

- browse the sample code and library files;
- run the Hugo Compiler on the platform of your choice, either through a graphical user interface or via the command line;
- view and set compile-time options such as switches, limits, and directories; and
- run a compiled Hugo file using the Hugo Engine.

Here's an example: on the author's machine, running under a Unix-like command line, the compiler executable **hc** is in a directory called **/boot/home/hugo**. The library files are in **/boot/home/hugo/lib**, and the source code for the game *Future Boy!* is in **/boot/home/hugo/fb**, with the main source file called **future.hug**.

It's possible to call the compiler to compile *Future Boy!* with a number of different options, including specifying the appropriate directories for source and library files, increasing the maximum possible number of routines, and printing all debugging information, the object tree, and statistics to a file. (Assume that the current directory is **/boot/home/hugo** and that none of the switches or directories are set in the source.)

Here's how that's done:

```
hc -lios $maxobjects=512 @source=fb @lib=lib future
```

(or

```
hc -lios '$maxobjects=512' '@source=fb', etc.
```

if the command shell requires that sequences beginning with '$' or '@' be contained in single-quotes or otherwise escaped). This makes use of various command-line options, including multiple switches, limit settings, and directory specifications. It sets the desired switches, changes the modifiable limit `MAXOBJECTS` from the compiler default, and points the compiler to look for source files in the **source** subdirectory and library files in the **lib** subdirectory (from the current directory).

II. A FIRST LOOK AT HUGO

II.a. Basic Concepts

There are a couple of basic concepts to become familiar with in order to begin working with Hugo. Once you begin to become familiar with them, you will hopefully be able to look at a chunk of Hugo source code and—even if you don't understand everything it's doing—be able to at least get a sense of the general organization.

First of all, the bulk of programming in Hugo will involve the creation of what are called *objects*. The word "object" in this sense has two meanings. First of all, in a programming sense, objects are discrete subsections of source code. They are referred to by individual names, and they "do something", whether that something is storing data or performing some set of functions or both. In the case of Hugo, however, these are not just abstract tools for structuring a program. Hugo objects are, more often than not, also representative of objects in the "physical world" of the game: people, places, and things. If, for example, you want to create a book in your game, you'll create a book object that may comprise the description of the book, what's written in it, how much it weighs, how many pages it has, what happens when you drop it, and anything else you choose to implement.

The rest of a Hugo program is mostly comprised of *routines*. These are the sections of code made up of commands or statements that facilitate the actual behavior of the program at different points in the story. (Routines can also be part of a containing object—we'll get to that in a little while.) Routines are less frequently (although more frequently in other languages) called "functions"—they may be thought of as performing an operation or series of operations, and then optionally *returning* some kind of answer or result. A program may have a routine called `DescribePlace` which, when invoked (or "called", in the parlance of programming) would print the description of a given location. The point of routines is that you don't have to repeat the same code every time you want a particular task done: you just have to call the routine. Write once, use many times.

The idea of return values from a routine is an important one and, while sometimes puzzling to novices, is actually quite uncomplicated. For instance, often a particular function will be described as "returning true" or "returning false"—all this means is that when it's done it returns either a non-zero value (usually 1) or a zero value, usually to indicate whether the function was successful or not at whatever it was being asked to do. A program will constantly be checking the return values of the routines it calls to determine if particular operations have been successful in order to decide what to do next. A routine can return any kind of value (listed shortly in *II.c Data Types*). A very simple example is a routine that performs a needed operation, such as adding two supplied values, *a* and *b*. Let's call it `AddTwoValues`. When `AddTwoValues` is called with the two supplied values, it will *return* the sum *a*+*b*.

For those familiar with the common programming languages such as C or Basic (including Visual Basic), Hugo will not be entirely visually unfamiliar. Individual objects and routines—as well as conditional blocks—are enclosed in braces as in C ("`{...}`"), but unlike C and other C-like languages, a semicolon is not required at the end of each line to tell the compiler when the line is finished, and the language itself is considerably less cryptic. Keywords, variables, routine and object names, and other tokens are not case-sensitive.

II.b. Hello, Sailor!

In the time-honored tradition of programming texts, the introduction to a new programming language is quite often a description of how to print the optimistic phrase "Hello, world" as an example of that particular language's form and substance. In the almost-equally time-honored tradition of interactive fiction, we'll start with the rallying cry "Hello, Sailor!". Here's how one accomplishes that in Hugo:

```
routine Main
{
     print "Hello, Sailor!"
     pause
     quit
}
```

The entire program consists of one routine. (Two routines are normally required for any Hugo program, the other being the `Init` routine, which is omitted in this simple example since there isn't anything required in the way of initialization.)

The `Main` routine is automatically called by the engine. It is from here that the central behavior of any Hugo program is controlled. In this case the task at hand is the printing of "Hello, Sailor!", followed by a wait for a keypress (the

`pause`) and an order to exit the program (i.e., `quit` it) so that we don't strand the program waiting for input from the player, which is the normal order of Hugo business.[4]

II.c. Data Types

Computer programs are mainly about two things: input and output (called *i/o*, for short), and modifying values. In fact, the bulk of a computer program (that is, what happens behind the scenes, whirring away, unbeknownst to the user) consists of setting, changing, and comparing various values. Hugo is no exception. All data in Hugo is represented in terms of 16-bit integers[5], treated as signed (-32768 to 32767) or unsigned (0 to 65535) as appropriate. It's up to the compiler and engine to decide what a particular value means in a given context. The name of any individual data type may contain up to 32 alphanumeric characters (as well as the underscore '_').

All of the following are valid data types:

Integer values `0, -10, 16800, -25005`
(constant values that appear in Hugo source code as numbers)

ASCII characters `'A', 'z', '7'`
(constant values equal to the common ASCII value for a character; i.e., 65 for 'A')

Objects `mysuitcase, emptyroom, player`
(constant values representing the object number of the given object)

Variables `a, b, score, TEXTCOLOR`
(changeable value-holders that may be set to equal another variable or constant value)

---

[4] Normally, unless the `Main` routine explicitly returns—as opposed to just running through to the closing brace—the Hugo Engine continues running. Those familiar with the C programming language may notice the slight difference here: whereas in C the `main()` function is the entry point for a C program, in Hugo `Init` is the entry point, and `Main` can be thought of as the "each-turn routine". For more elaboration on the execution pattern of a Hugo program, see *IX. THE GAME LOOP*.

[5] While it's a little beyond the scope of this manual to talk about what exactly a 16-bit integer is (partly because you don't need to worry about it, other than to know they involve a range of 65536, either 0 to 65535 or -32768 to 32767). Essentially, "bits" refer to 1s or 0s in a base 2 number system (so that the right-most bit is the 1s, the next-to-right-most is the 2s, the next the 4s, the next the 8s, etc.) For example, the 4-bit number 1100 is equal to decimal 12, since 8+4=12. (If you're familiar with bitwise notation, you already knew that. If you're not, it probably didn't particularly clear anything up, but as always, not to worry.)

Constants `true, false, BANNER`
(constant—obviously—values that are given a name similarly to a variable, but are non-modifiable)

Dictionary entries `"a", "the", "basketball"`
*(The appearance of "*the*" in a line of code actually refers to the location in the dictionary table where the word "*the*" is stored. Dictionary entries are non-modifiable.)*

Array elements `ranking[1]`
(a series of one or more changeable values that may be referenced from a common base point)

Array addresses `ranking`
(the base point of an array—see above; the array address itself is non-modifiable, unlike the contents of the array)

Properties `nouns, short_desc, found_in`
(variable attachments of data relating specifically to objects)

Attributes `open, light, transparent`
(less complex attachments of data describing an object, which may be specified as either having or not having the given attribute)

Most of these types are relatively straightforward, representing in most cases a simple value. As noted, some values are dynamic (modifiable), while others are static (non-modifiable). Dictionary entries are addresses in the dictionary table (comprising all dictionary words in the .HEX file), with the empty string "" having the value 0. Array addresses (as opposed to separate array elements) represent the address at which the array begins in the array table (comprising all array data in the .HEX file). Properties and attributes treated as discrete values represent the number of that property or attribute, assigned sequentially as the individual property or attribute is defined.

As mentioned, routines also return values, as do built-in[6] engine functions, so that

```
FindLight(room)
```

[6] Built-in functions are just like other Hugo functions except that they're never defined anywhere in source code or any library file: the compiler and engine always know about them. To distinguish them, they're generally printed in lowercase, whereas program-defined routines (including library routines) are almost always capitalized.

and

```
parent(object)
```

are also valid integer data types.[7]

It's good medicine to be as descriptive as possible in naming symbols, regardless of what you're naming. A variable that holds the count of a number of objects *could* be called `n`, but it's almost always better (especially after the fact, when you're looking at code you've written days or even months before) to call it something more helpful like `object_count`.

At this point it's probably helpful to know that you can assign a value to a variable using the form:

```
<some variable> = <some value>
```
[8]

For instance, to set the variable `x` equal to 5, you would use:

```
x = 5
```

To set it equal to element 4 of array `some_array`, you would use:

```
x = some_array[4]
```

Note: What follows is one of those if-you-don't-quite-understand-yet-don't-panic sections of the manual: unless you can think of a place off the top of your head where something like this would be useful, it'll probably be a little while until you need to use it.

When you want to get the return value of a routine, you would use:

```
x = Routine
```

If, then, you ever need to get the indexed address of a routine to use it as a value, as you may at some point, you obviously won't be able to do:

```
x = Routine
```

---

[7] Routine addresses are also stored as 16-bit integers. However, those versed at all in such calculations will notice that if such a value was treated as an absolute address, then any addressable executable code would be limited to 64K in size (65536 bytes, the maximum size of an unsigned 16-bit integer). Such is not the case, since the routine address is actually an indexed representation of the absolute address, allowing Hugo games to far exceed any such limit in their size of executable code.

[8] The section *IV.e. Operators and Assignments* goes into greater detail on assigning values to variables.

again and hope that this time it will assign the address of `Routine` to the variable `x`, since that will assign to `x` the value *returned by* `Routine`. Instead, you can use the address operator '&', as in:

```
x = &Routine
```

which won't actually call `Routine` but will instead only assign the routine's address to `x` (or, as we'll see later,

```
x = &object.property
```

to get a property routine address instead of calling the property routine itself.)

II.d. Multiple Lines

If any single command is too long to fit on one line, it may be split across several lines by ending all but the last with the control character '\'.

```
"This is an example string."
```

and

```
x = 5 + 6 * higher(a, b)
```

are the same as

```
"This is an example \
string."
```

and

```
x = 5 + 6 * \
     higher(a, b)
```

String constants, such as in the below `print` statement, are an exception in that they do not require the '\' character at the end of each line (although, as shown just above, it's not wrong to use it).

```
print "The engine will properly
     print this text, assuming a
     single space at the end of each
     line."
```

will result in:

```
The engine will properly print this text, assuming a
single space at the end of each line.
```

Care must be taken, however, to ensure that the closing quotes are not left off the string constant. Failing that, the compiler will likely generate a “Closing brace missing” or similar error when it overruns the object/routine/event boundary looking for a resolution to the odd number of quotation marks.

(Habitual double-space-after-a-period typists may find it useful to use the ‘\’ character for line continuation in situations like this:

```
print "Here, we'll end a sentence on one line.  \
     However, we'd like to make sure there
     are two spaces before the second sentence."
```

giving:

```
Here, we'll end a sentence on one line.  However, we'd
like to make sure there are two spaces before the
second sentence.
```

since normally, if the ‘/’ were omitted after “...on one line.”, the compiler would assume only a single space before continuing with “However...” from the next line.)

Also, most lines ending in a comma, `and`, or `or` will automatically continue on to the next line (if they occur in a line of code). In other words:

```
x[0] = 1, 2, 3,      ! array assignment x[0]..x[4]
     4, 5
```

and

```
if a = 5 and
     b = "tall"
```

get compiled the same as:

```
x[0] = 1, 2, 3, 4, 5
```

and

```
if a = 5 and b = "tall"
```

This is provided primarily so that lengthy lines and complex expressions do not have to run off the right-hand side of the screen during editing, nor do they continually need to be extended using '\' and the end of each line.

Note: Multiple lines that are not strictly code, such as property assignments in object definitions—to be discussed shortly—must still be joined with '\', as in

```
nouns "plant", "flower", "marigold", \
     "fauna", "greenery"
```

and similar cases, even if they end in a comma.

There is a complement to the '\' line-control character: the ':' character allows multiple lines to be put together on a single line, i.e.:

```
x = 5 : y = 1
```

or

```
if i = 1:  print "Less than three."
```

Which the compiler translates to:

```
x = 5
y = 1
```

and

```
if i = 1
     {print "Less than three."}
```

(We'll get to exactly what that "`if...print...`" business means in just a little bit in *IV.h Conditional Expressions and Program Flow*.)

II.e. Comments

Comments allow you to insert notes into source code to serve as reminders, descriptions of what a particular chunk of code does, put a curse upon the libary/language author, or whatever else you want. Comments are

very helpful, and beginning programmers tend to put in either too many comments or too few. Despite the complaints that some people may have about over-commented code—generally referring to commenting a line like:

```
x = 5
```

with the rather obvious explanation of "set x equal to 5"—it's always better to err on the side of too many comments in order to avoid the situation that every programmer find himself or herself in at least once (and once only if very, very lucky) of trying to remember what a piece of code does that you wrote yesterday, or last week, or several months ago. Comment, comment, comment.[9]

There are two types of comments. Comments on a single line begin with a '!'. Anything following on the line is ignored. Multiple-line comments are begun with '!\' and ended with '\!'.

```
! A comment on a single line

!\ A multiple-line
   comment \!
```

Note: The '!\' combination must come at the start of a line to be significant; it cannot be preceded by any other statements or remarks. Similarly, the '\!' combination must come at the end of a line (or alone on an otherwise blank line).

II.f. Compiler Errors And Warnings

The compiler is pretty good about catching you when you do something that isn't going to work. When it encounters something in your source code that doesn't make sense, or is illegal in terms of the Hugo language, it'll tell you.

A compiler error is generally of one of two types. A fatal error looks like this:

```
Fatal error:  <message>
```

and halts compiler execution. Fatal errors include things like not being able to find a requested file, encountering some sort of i/o difficulty (such as not being

[9] But keep an eye out for issues of comment maintenance. Again, a good comment should add clarity to a section of code, but it (usually) shouldn't restate exactly what the code is doing. Doing that just means that when you change the code, you have to change the comment to keep it accurate, too, which if you've overcommented means doing the same thing twice, and increasing the chances of getting out of sync so that the comment doesn't perfectly reflect the code it's supposed to be commenting.

able to read from or write to a necessary file), or having encountered something in the source code that makes it impossible to continue with compilation.

A non-fatal error typically looks like:

```
<filename>:<line>:  Error:  <message>
```

Non-fatal errors are usually programming mistakes: either doing something illegal (like trying to assign a value to something to which you're not allowed to assign a value), making a syntax error such as using a symbol name that the compiler doesn't know about (often due to a typing mistake), or making a formatting mistake (like missing something that the compiler knows is supposed to be coming next but you forgot to include). Unless the -a switch is specified at invocation to tell the compiler to quit after the first error, multiple non-fatal errors may be printed. The side-effect of this is that a specific error (particularly a formatting error) may affect many lines of code after it, so that the compiler—having become lost and not really knowing what you're trying to do—may report a whole string of errors, even on lines that, if the compiler understood their proper context, would be error free.[10]

When a compiler issues a warning, it looks like:

```
<filename>:<line>:  Warning:  <message>
```

Compilation will continue, but this is an indication that the compiler suspects a problem at compile-time.

If the -e switch has been set during invocation to generate expanded-format errors, error output looks like:

```
<FILENAME>:  <LOCATION>
(Error-causing line)
"ERROR:  <error message>"
```

It prints the section of code that caused the error, followed by an explanation of the problem. Compilation will generally continue unless the -a switch has been set.

Note: The section of offending code may not be printed exactly as it appears in the source when using the -e switch, since the compiler occasionally mildly paraphrases and rebuilds the source line into a more rigid format before finally compiling it.

[10] Which is why, in certain cases, the -a switch can be helpful.

II.g. Compiler Directives

A number of special commands may be used that aren't really part of a Huge program *per se*, but rather give instructions to the compiler itself to determine (a) how the source code—or a part thereof—is read by the compiler and (b) what special output will be generated at compile-time. These special commands or instructions are called *compiler directives*, and are preceded with a '#' character to set them apart.

To set switches within the source code so that they do not have to be specified each time the compiler is invoked for that particular program, the line

```
#switches -<sequence>
```

will set the switches specified by <sequence>, where <sequence> is a string of characters representing valid switches, without any separators between characters. Many programmers may find it useful to make

```
#switches -ls
```

the first line in every new program, which will automatically print a statistical summary of compilation (plus any warnings or errors) to the **.lst** list file.

Using

```
#version <version>[.<revision>]
```

specifies that the file is to be used with version <version>.<revision> of the compiler. If the file and compiler version are mismatched, a warning will be issued.

Note: The #version directive is intended mainly for things like library files, and although you may use it in your own source files, it isn't necessary. Its general usage is largely deprecated.

To include the contents of another file at the specified point in the current file, use

```
#include "<filename>"
```

where **<filename>** is the full path and name of the file to be read. When **<filename>** has been read completely, the compiler resumes with the statement immediately following the #include directive. There is no limit on the number

of files that a single file may include; also, a file may include a file which includes another file which includes another file and so on. (A file or set of files can be compiled into a precompiled header using the -h switch, and then linked using `#link` instead of `#include`. See *APPENDIX E: PRECOMPILED HEADERS*.)

A useful tool for managing Hugo source code is the ability to use compiler flags for conditional compilation. A compiler flag is simply a user-defined marker that can control which sections of the source code are compiled. In this way, a programmer can demarcate sections of a program that can be included or excluded at will. For example, the library files **hugolib.h**, **verblib.h**, and **verblib.g** check to see if a flag called `DEBUG` has been set previously (as it is in **sample.hug**). Only if it has do they include the **hugofix.h** and **hugofix.g** files, which in turn provide certain debugging features to a running Hugo program. (For a final version to be released to the general public for playing, then, by simply not setting the `DEBUG` flag those special features are not enabled.)

To set and clear flags, use

```
#set <flagname>
```

and

```
#clear <flagname>
```

respectively. (Flags can also be explicitly set on the command line during compiler invocation via

```
hc #<flagname> <sourcefile>...
```

similarly to compiler limit settings and directories, with the same caveat that on some systems it may be necessary to enclose `#<flagname>` in single quotes or otherwise escape it, if required.)

Then, check to see if a flag is set or not (and include or exclude the specified block of source code) by using

```
#ifset <flagname>
        ...conditional block of code...
#endif
```

or

```
#ifclear <flagname>
        ...conditional block of code...
#endif
```

Conditional compilation constructions may be nested up to 32 levels deep. (Note also that compiler flags can be specified in the invocation line as **`#<flag name>`**.)

"`#if set`" and "`#if clear`" are the long form of "`#ifset`" and "`#ifclear`", allowing usage of "`#elseif`" for code such as:

```
#set THIS_FLAG
#set THAT_FLAG

#if clear THIS_FLAG
#message "This will never be printed."
#elseif set THAT_FLAG
#message "This will always be printed."
#else
#message "But not this if THAT_FLAG is set."
#endif
```

Use "`#if defined <symbol>`" and "`#if undefined <symbol>`" to test if objects, properties, routines, etc. have previously been defined, where `<symbol>` is the name of the object, property, routine, etc. in question.

As seen above, the `#message` directive can be used as

```
#message "<text>"
```

to output `<text>` when (or if) that statement is processed during the first compilation pass.

Including "`error`" or "`warning`" before "`<text>`" as in

```
#message error "<text>"
```

or

```
#message warning "<text>"
```

will force the compiler to issue an error or warning, respectively, as it prints "`<text>`".

Note: It's worth pointing out that all of the text printed in the above `#if/#elseif` example is *compile-time* output, not *runtime* output. That is, it's printed only when the compiler initially compiles the source code, not when a player plays the actual game.

It is also possible to include inline limit settings, such as

```
$<setting>=<limit>
```

in the same way as in the invocation line. However, an error will be issued if, for example, an attempt is made to reset `MAXOBJECTS` if one or more objects have already been defined. Any limit settings in the code of a program must be done before the particular data type for which a new limit is being set has been used.

II.h. What Should I Be Able To Do Now?

By now you should:

- be able to look at Hugo source code and start to see the separation into different discrete parts, such as routines;
- have a general idea about the various Hugo data types, and be able to differentiate them in Hugo source code;
- know about different aspects of Hugo source code formatting such as multiple lines and comments;
- know how to read an error produced by the Hugo Compiler; and
- know how to use inline compiler directives to set switches, flags, limits, and directories.

To experiment a little, make a copy of **`sample.hug`** and call it something like **`test.hug`** so that we can modify and use it without changing the original sample game source code. Pick a line in the new file **`test.hug`** like:

```
#set DEBUG
```

and add some garbage letters to change it to

```
asdf#set DEBUG
```

Now, when you compile, you'll see:

```
test.hug:12:  Error:  Unknown compiler directive:
asdf
```

(Depending on the contents of **`test.hug`**, the actual line number may vary.) Once we've seen the effect of that, go back and remove the "asdf" from **`test.hug`**. Next, let's try adding the line:

```
$MAXOBJECTS=50
```

to the start of **test.hug**. Compile again, and you'll see this time a whole bunch of compiler errors. Most importantly are the first couple, which look something like:

```
test.hug:691:  Error:  Maximum of 50 objects exceeded
```

(The other errors basically follow from the last few objects in **test.hug** not getting defined, and the compiler subsequently knowing that a particular symbol is the name of an object.)

Feel free to experiment with **test.hug** by adding comments, changing lines, commenting out various objects or routines or other sections of codes, and seeing what happens when you try to compile it and run it.

III. OBJECTS

III.a. Getting To Know Your Objects

Objects are the building blocks of any Hugo program. Anything that will be accessible to a player during the game—including items, rooms, other characters, and even directions—will most likely be defined as an object. The basic object definition looks like this:

```
object <objectname> "object name"
{
      ...
}
```

For example, a suitcase object might be defined as:

```
object mysuitcase "suitcase"
{}
```

The enclosing braces are needed even if the object definition has no content (yet). The only data attached to the suitcase object are—from right to left—a name ("suitcase"), an internal identifier (mysuitcase), and membership in the basic object class.

The compiler assigns the object labeled <objectname> the next sequential object number. The first-defined object is object 0; the next-defined object is object number 1; the one after that is 2, etc. This is academic, however, as a programmer almost never need know what object number a particular object is—except for certain debugging situations—and can always refer to an object by its label <objectname>. If no explicit "object name" (or name property) is provided, the compiler automatically gives it the name "(<objectname>)", i.e., <objectname> in parentheses. That is, whereas

```
object mysuitcase "suitcase"
{}
```

creates an object called "suitcase",

```
object placeholder
{}
```

creates an object called "(placeholder)". Usually the latter is used for system objects or classes (see *III.e Classes*) that will never actually appear in a game.

Note: The compiler automatically creates an object called "display" as the last-defined object if no other object named "display" is defined by the program (or the library). The display object can be used to get information about the engine's output state and capabilities. See section *XI.a The Display Object*.

III.b. The Object Tree

In order for objects to have a "physical place" in the game, i.e., to be in a room or contained in another object or beside another object, they must occupy a position in the object tree. The object tree is a simple map which represents the relationships between all objects in the game. The total number of objects is held in the global variable `objects`.

The "nothing" object is defined in the library as object 0 and is referred to in code using the label `nothing`. This is the root of the object tree, upon which all other objects are based.[11] (And again, the name "nothing" is given to this first object by the library.)

Note: When using the standard library routines, ensure that no objects (or classes, to be discussed later) are defined before `hugolib.h` is included. Problems will arise if the first-defined object—object 0—is not the `nothing` object. Currently the library will point this out for you as a runtime error if for some reason it's not the case.

When referring to object numbers, this manual is generally referring to the name given the object in the source code: i.e., `<objectname>`. The compiler automatically assigns each object an object number, and refers to it whenever a specified `<objectname>` is encountered.

[11] It's also no coincidence that the "nothing" object is equal in its value to 0, which also represents the empty string "" (see *II.c Data Types*). The fact that these three are (value-wise, at least) identical will come in handy, as what it means in practice is that 0/null/empty/nothing/etc. is the same in every context.

Here is an example of an object tree:

```
nothing
|
Room
|
Table———Chair———Book———Player
|               |
Bowl            Bookmark
|
Spoon
```

A number of built-in functions can be used to read the object tree.

```
parent
sibling
child
youngest
elder
eldest      (same as child)
younger     (same as sibling)
```

and

```
children
```

Each function takes a single object as its argument, so that

```
parent(Table)     = Room
parent(Bookmark)  = Book
parent(Player)    = Room
child(Bowl)       = Spoon
child(Room)       = Table
child(Chair)      = 0   ("nothing")
sibling(Table)    = Chair

sibling(Player)   = 0   ("nothing")
youngest(Room)    = Player
youngest(Spoon)   = 0   ("nothing")
elder(Chair)      = Table
elder(Table)      = 0   ("nothing")
```

and

```
children(Room)    = 4
children(Table)   = 1
children(Chair)   = 0
```

(In keeping with the above explanation of object numbers and `<objectname>`, the functions in the first set actually return an integer number that refers to the object `<objectname>`.)

To better understand how the object tree represents the physical world, the table, the chair, the book, and the player are all in the room. The bookmark is in the book. The bowl is on the table, and the spoon is on the bowl. The Hugo library will assume that the player object in the example is standing; if the player were seated, the object tree might look like:

```
nothing
|
Room
|
Table———Chair———Book
|       |       |
Bowl    Player  Bookmark
|
Spoon
```

and

```
child(Chair)      = Player
parent(Player)    = Chair
children(Chair)   = 1
```

(Try compiling **`sample.hug`** with the -o switch in order to see the object tree for the sample game. Or, if the `DEBUG` flag was set during compilation, use the HugoFix[12] command "`$ot`" or "`$ot <object>`" during play to view the current state of the object tree during play. Compiling with the -d switch will generate a debuggable (.HDX) version of the file—the object tree can then be viewed directly from the debugger.)

To initially place an object in the object tree, use

```
in <parent>
```

in the object definition, or, alternatively

[12] See *APPENDIX D: HUGOFIX AND THE HUGO DEBUGGER*.

```
nearby <object>
```

or simply

```
nearby
```

to give the object the same parent as `<object>` or, if `<object>` is not specified, the same parent as the last-defined object. If no such specification is given (i.e., if you don't tell the compiler explicitly where to place the new object), the parent object defaults to 0—the "nothing" object as defined in the library. All normal room objects have 0 as their parent.

Therefore, the expanded basic case of an object definition is

```
object <objectname> "object name"
{
     in <parent object>
     ...
}
```

(Ensure that the opening brace '{' does not come on the same line as the `object` definition. Trying to do:

```
object <objectname> "object name" {...
```

is not permitted.)

The table in the example presumably had a definition like

```
object table "Table"
{
     in room
     ...
}
```

To put the suitcase object defined earlier into the empty room in **`shell.hug`**:

```
object mysuitcase "suitcase"
{
     in emptyroom
}
```

Objects can later be moved around the object tree using the `move` command as in:

```
move <object> to <new parent>
```

which, essentially, disengages `<object>` from its old parent, makes the sibling of `<object>` the sibling of `<object>`'s elder, and moves `<object>` (along with all its possessions) to the new parent.

Therefore, in the original example, the command

```
move bowl to player
```

would result in altering the object tree to this:

```
nothing
|
Room
|
Table———Chair——-Book———Player
                |         |
                Bookmark  Bowl
                          |
                          Spoon
```

There is also a command to remove an object from its position in the tree:

```
remove <object>
```

which is the same as

```
move <object> to 0
```

The object may of course be moved to any position later.

Logical tests can also be evaluated with regard to objects and children. The structure

```
<object> [not] in <parent>
```

will be true if `<object>` is in `<parent>` (or false if `not` is used). In this way, you can write a piece of code that looks something like:

```
if mysuitcase in bedroom
{
     "The suitcase is in the bedroom."
}
else
```

```
{
     print "The suitcase is not in the bedroom."
}
```

(We'll cover the "`if...else...`" structure in *IV.h Conditional Expressions and Program Flow*.)

III.c. Attributes

Attributes are essentially qualities that every object either does or doesn't have[13]. An attribute is defined as

```
attribute <attribute name>
```

Up to 128 attributes may be defined. Those defined in **`hugolib.h`** include:

`known`	if an object is known to the player
`moved`	if an object has been moved
`visited`	if a room has been visited
`static`	if an object cannot be taken
`plural`	for plural objects (i.e., some hats)
`living`	if an object is a character
`female`	if a character is female
`openable`	if an object can be opened
`open`	if it is open
`lockable`	if an object can be locked
`locked`	if it is locked
`unfriendly`	if a character is unfriendly
`light`	if an object is or provides light
`readable`	if an object can be read
`switchable`	if an object can be turned on or off
`switchedon`	if it is on
`clothing`	for objects that can be worn
`worn`	if the object is being worn
`mobile`	if the object can be rolled, etc.
`enterable`	if an object is enterable
`container`	if an object can hold other objects
`platform`	if other objects can be placed on it[14]

[13] For this reason, attributes are sometimes thought of as being "lightweight classes" in that, as can be seen in the list of attributes, they generally categorize an object as a certain "kind" of object—although other than flagging the object with that particular quality they have no other direct effect.

`hidden`	if an object is not to be listed
`quiet`	if container or platform is quiet (i.e., the initial listing of contents is suppressed)
`transparent`	if object is not opaque
`already_listed`	if object has been pre-listed (i.e., before a `WhatsIn` listing[15])
`workflag`	for system use
`special`	for miscellaneous use

Some of these attributes are actually the same attribute with different names. This is primarily just to save on the absolute number of attributes defined and is accomplished via

```
attribute <attribute2> alias <attribute1>
```

where <`attribute1`> has already been defined. For example, the library equates visited with moved (since, presumably, they will never apply to the same object—rooms are never moved and objects are never visited), so:

```
attribute visited alias moved
```

In this case, an object which is visited is also, by default, moved, so it is expected that attributes which are aliased will never both need to be checked under the same circumstances. For the most part, you should never need to alias your own attributes, although it's helpful to know what it means since the library does it, and you may run across it in other places.

Attributes are given to an object during its definition as follows:

```
object <objectname> "object name"
{
     is [not] <attribute1>, [not] <attribute2>, ...
     ...
}
```

Note: The `not` keyword in the object definition is important when using a class instead of the basic object definition, where the class may have predefined attributes that are undesirable for the current object.

---

[14] The container and platform attributes are mutually exclusive. An object cannot have both attributes, since in the library the idea of containment is one of an object being either "in" *or* "on" another object. There are available classes that aren't part of the standard library distribution that allow an object to function as both.

[15] `WhatsIn` is a library function used to list in formatted fashion all the objects present in a location: see *APPENDIX B: THE HUGO LIBRARY.*

To give the suitcase object some appropriate attributes at compile-time, expand the object definition to include

```
object mysuitcase "suitcase"
{
     in emptyroom
     is openable, not open
     ...
}
```

Even if an object was not given a particular attribute in its object definition, it may be given that attribute at any later point in the program with the command

```
<object> is [not] <attribute>
```

where the `not` keyword clears the attribute instead of setting it. For example, when the suitcase is opened, somewhere (likely in the library), the command

```
mysuitcase is open
```

will be executed. When the suitcase is closed, the command will be:

```
mysuitcase is not open
```

Attributes can also be read using the `is` and `is not` structures and evaluate to either true or false. In code, the expression

```
<object> is [not] <attribute>
```

returns true (1) if `<object>` is (or is not, if `not` is specified) `<attribute>`. Otherwise, it returns false (0). Therefore, given the suitcase object definition:

```
object mysuitcase "suitcase"
{
     in emptyroom
     is openable, not open
     ...
}
```

the following equations hold true:

```
mysuitcase is openable = 1    ! or true
```

```
mysuitcase is open = 0         ! or false
mysuitcase is locked = 0       ! or false
```

III.d. Properties

Properties are considerably more complex than attributes. First, not every object may have every property; in order for an object to have a property, it must be specified in the object definition at the time you create the object. As well, properties are not simple on/off flags. They are sets of valid data associated with an object, where the values may represent almost anything, including object numbers, dictionary addresses, integer values, and sections of executable code.

These are some valid properties as they would appear in an object definition (using property names defined in **hugolib.h**)[16]:

```
nouns "tree", "bush", "shrub", "plant"

size 20

found_in livingroom, entrancehall

long_desc
{
     "Exits lead north and west.  A door is set
     in the southeast wall."
}

short_desc
{
     "There is a box here.  It is ";
     if self is open
          print "open";
     else
          print "closed";
     print "."
}

before[17]
{
```

[16] Don't worry too much about the specifics about what this code is supposed to be doing, or about the details of the language syntax. We'll cover all of that in due course.
[17] Just for clarity: the `Art` routine from **hugolib.h** prints the appropriate article, if any, followed by the name of the object, such as "an apple" or "a suitcase". The `Acquire` routine returns true only if the first object's `holding` property plus the `size` property of the second object does not exceed the `capacity` property of the first object (i.e., if there's room in the first object to move the second object into it).

```
		object DoGet
		{
			if Acquire(player, self)
			{
				"You pick up ";
				print Art(self); "."
			}
			else
				return false
		}
	}
```

The `nouns` property contains four dictionary addresses; the `size` property is a single integer value; the `found_in` property holds two object numbers; and the long and short description properties are both *property routines*, which instead of just containing one or more simple values stored as a data type are actually sections of executable code attached to the object.

The `before` property is a special case. This *complex property routine* is defined by the compiler and handled differently by the engine than a normal property routine. In this case, the property value representing the routine address is only returned if the global variables `object` and `verbroutine` contain the object in question and the address of the `DoGet` routine, respectively. If there is a match, the routine is executed before `DoGet`, which is the library routine (in **`verblib.h`**) that normally handles the taking of objects. (There is also a companion to `before` called `after`, which is checked after the verb routine has been called.) See *V.c Before And After Routines* for further elucidation.

There will be more on property routines and complex property routines later. For now, think of a property as simply containing one or more values of some kind.

A property is defined similiarly to an attribute as

```
property <property name>
```

A default value may be defined for the property using

```
property <property name> <default value>
```

where `<default value>` is a constant or dictionary word. For objects without a given property, attempting to find that property will result in the default value. If no default is explicitly declared, it is 0 (or “” or the “nothing” object, whatever is appropriate in context—since they all represent the same zero value).

The list of properties defined in **`hugolib.h`** is:

`name`	the basic object name
`before`	pre-verb routines
`after`	post-verb routines
`noun`	noun(s) for referring to object
`adjective`	adjective(s) for describing object
`article`	“a”, “an”, “the”, “some”, etc.
`preposition`	“in”, “inside”, “outside of”, etc.
`pronoun`	appropriate for the object in question
`react_before`	to allow reaction by an object that is not
`react_after`	directly involved in the action
`short_desc`	basic “X is here” description
`initial_desc`	supersedes `short_desc` (or `long_desc` for locations)
`long_desc`	detailed description
`found_in`	in case of multiple locations (virtual, *not* physical parent objects[18])
`type`	to identify the type of object
`size`	for holding/inventory
`capacity`	“ “ “
`holding`	“ “ “
`reach`	for limiting object accessibility
`list_contents`	for overriding normal listing
`in_scope`	actor(s) that can access an object
`parse_rank`	for differentiating like-named objects
`exclude_from_all`	for interpreting “all” in player input
`door_to`	for handling “`>ENTER <object>`“
`n_to`	
`ne_to`	
`e_to`	
`se_to`	
`s_to`	
`sw_to`	(for rooms only, where an exit leads)
`w_to`	
`nw_to`	
`u_to`	
`d_to`	
`in_to`	
`out_to`	
`cant_go`	message if a direction is invalid
`extra_scenery`	unimportant words/objects in location desc.

---

[18] In this usage, a “physical” parent is one in the object tree. The `found_in` property allows you have an object considered in a location (i.e., a room object) without it being “physically” in that room object.

`each_turn`	a routine called each turn
`key_object`	if `lockable`, the proper key
`when_open`	supersedes `short_desc`
`when_closed`	" "
`ignore_response`	for characters
`order_response`	" "
`contains_desc`	instead of basic "Inside X are..."
`inv_desc`	for special inventory descriptions
`desc_detail`	parenthetical detail for object listing
`misc`	for miscellaneous use

(For a detailed description of how each property is used, see *APPENDIX B: THE HUGO LIBRARY*.)

The following properties are also defined and used exclusively by the display object:

`screenwidth`	width of the display, in characters
`screenheight`	height of the display, in characters
`linelength`	width of the current text window
`windowlines`	height of the current text window
`cursor_column`	horizontal and vertical position of
`cursor_row`	the cursor in the current text window
`hasgraphics`	true if the current display is graphics-capable
`title_caption`	dictionary entry giving the full proper name of the program (optional)
`statusline_height`	of the last-printed status line

Property names may be aliased similarly to attributes using:

```
property <property2> alias <property1>
```

where `<property1>` has already been defined. The library aliases (among others) the following:

```
nouns alias noun
adjectives alias adjective
prep alias preposition
pronouns alias pronoun
```

Whereas a simple property is expressed as

```
<object>.<property>
```

The number of elements to a property with more than a single value can be found via

```
<object>.#<property>
```

and a single element is expressed as

```
<object>.<property> #<element number>
```

Note: `<object>.<property>` is simply the shortened version of `<object>.<property> #1`.

To add some properties to the suitcase object, expand the object definition to:

```
object mysuitcase "big green suitcase"
{
     in emptyroom              ! done earlier
     is openable, not open     !

     nouns "suitcase", "case", "luggage"
     adjective "big", "green", "suit"
     article "a"
     size 25
     capacity 100
}
```

Based on the parser's rules for object identification, the suitcase object may now be referred to by the player as "big green suitcase", "big case", or "green suitcase" among other combinations. Even "big green" and "suit" may be valid, provided that these don't also refer to other objects within valid scope such as "a big green apple" or "your suit jacket".

The basic form for identification by the parser is

```
<adjective 1> <adj. 2> <adj. 3>...<adj. n> <noun>
```

where any subset of these elements is allowable. However, the noun must come last, and only one noun is recognized, so that

```
<noun> <noun>
```

and

```
<noun> <adjective>
```

as in “luggage case” and “suitcase green” are not recognized.

One occasional source of befuddling code that doesn’t behave the way the programmer intended is not allowing enough slots for a property on a given object. That is, if an object is originally defined with the property

```
found_in kitchen
```

and later, the program tries to set

```
<object>.found_in #2 = livingroom
```

in order to make the object available in both the kitchen *and* the living room, it will have no substantial effect. That is, there will be no space initialized in `<object>`‘s property table for a second value under `found_in`. Trying to read `<object>.found_in #2` will return a value of 0—a non-existent property—not the number of the `livingroom` object.

To overcome this, if it is known that eventually a second (or third, or fourth, or ninth) value is going to be set—even if only one value is defined at the outset—use

```
found_in kitchen, 0[, 0, 0,...]
```

in the object definition. (A useful shortcut for initializing multiple zero values is to use

```
found_in #4
```

instead of

```
found_in 0, 0, 0, 0
```

where #*n* initializes *n* zero values in the object definition.)

As might be expected, combinations of properties are read left-to-right, so that

```
location.n_to.name
```

is understood as

```
(location.n_to).name
```

which is, in other words, the `name` property of the object stored in `location.n_to`.

III.e. Classes

Classes are objects that are specifically intended to be used as prototypes for one or more similar objects. They're extremely useful for when you want to create a number of objects that will all share certain basic characteristics. Here is how a class is defined:

```
class <classname> ["<optional name>"]
{
     ...
}
```

with the body of the definition being the same as that for an object definition, where the properties and attributes defined are to be the same for all members of the class.

For example:

```
class box
{
     noun "box"
     long_desc
          "It looks like a regular old box."
     is openable, not open
}

box largebox "large box"
{
     article "a"
     adjectives "big", "large"
     is open
}

box greenbox "green box"
{
     article "a"
     adjective "green"
     long_desc
```

```
                "It looks like a regular old box,
                only green."
        }
```

(Beginning the `long_desc` property routine on the line below the property name is understood by the compiler as:

```
        long_desc
        {
                "It looks like a regular old box,
                only green."
        }
```

Since the property is only one line—a single line of text to print—the braces are unnecessary.)

The definition of an object derived from a particular class is begun with the name of the prototype object instead of `object`. All properties and attributes of the class are inherited (except for its position in the object tree), unless they have been explicitly defined in the new object (in which case they take precedence over any defaults defined in the class).

That is, although the box class is defined without the open attribute, the `largebox` object will begin the game as open, since this is in the `largebox` definition. It will begin the game as `openable`, as well, as this is inherited from the `box` class.

And while the `largebox` object will have the `long_desc` of the box class, the `greenbox` object replaces the default property routine with a new description. (An exception to this is an "`$additive`" property, to be discussed later, where new properties are added to those of previous classes.)

It is also possible to define an object using a previous object as a class even though the previous object was not explicitly defined as a class (using the `class` keyword). Therefore,

```
largebox largeredbox "large red box"
{
        adjectives "big", "large", "red"
}
```

is perfectly valid. We created what amounts to a "copy" of `largebox`, with a different name ("large red box" this time) and a different set of adjectives to refer to it.

Occasionally, it may be necessary to have an object or class inherit from more than one previously defined class. This can be done using the "`inherits`" instruction.

```
<class1> <objectname> "name"
{
     inherits <class2>[, <class3>,...]
     ...
}
```

or even

```
object <objectname> "name"
{
     inherits <class1>, <class2>[, <class3>,...]
     ...
}
```

The precedence of inheritance is in the order of occurrence. In either example, the object inherits its properties and attributes first from `<class1>`, then from `<class2>`, and so on.

The Hugo Object Library (**`objlib.h`**) contains a number of useful class definitions for things like rooms, characters, scenery, vehicles, etc. Sometimes, however, it may be desirable to use a different definition for, say, the room class while keeping all the others in the Object Library.

Instead of actually editing **`objlib.h`**[19], use:

```
replace <class> ["<optional name>"]
{
     (...completely new object definition...)
}
```

where `<class>` is the name of a previously defined object or class, such as "`room`". All subsequent references to `<class>` will use this object instead of the previously defined one. (Note that this means that the replacement must come *before*[20] any uses of the class as the parent class for other objects.)

---

[19] Editing the library files is generally not recommended, and not only because you'll have to re-apply your changes if you update to a newer release of the library. If you absolutely must change one of the library files, make a copy first.

[20] In terms of order-of-inclusion.

III.f. What Should I Be Able To Do Now?

By now you should:

- be able to create simple objects and add them to an existing game—whether an empty game based on **`shell.hug`** or a copy of **`sample.hug`** complete with existing objects and locations;

- experiment by adding new objects, giving them different names and starting locations as well as nouns and adjectives to describe them, assigning new property values or modifying existing ones, setting different attributes, etc.;

- have a basic understanding of how the object tree works in terms of how objects are arranged within the physical world of the game, including rooms or locations, objects within those locations, and objects within other objects.

IV. HUGO PROGRAMMING

IV.a. Variables

What is a variable, exactly? Let's start with the difference between a constant value and a variable value. The number 6 is a constant: we can't change it. We can't tell the program: "In this particular circumstance, let's treat this 6 like it was actually 21." Consider a situation, however, where we may want to record a particular value at one point in order to refer to it later. In other words, we may want to use a value that we won't know at the time we write the code that will be using it.

Here's a piece of code that, as we'll see shortly, prints a single line of output with a number in the middle:

```
print "The temperature is "; number temp; " degrees."
```

That statement may print

```
The temperature is 10 degrees.
```

or

```
The temperature is -9 degrees.
```

or any other similar variation depending on what the *variable* `temp` happens to be equal to at the time.[21]

Hugo supports two kinds of variables: *global* and *local*. Either type simply holds an integer value, so a variable can hold a simple value, an object number, a dictionary address, a routine address, or any other standard Hugo data type through an assignment such as:

[21] Those readers who weren't already aware of variables and their usage may at this point be starting to have high-school algebra flashbacks. That's because we're talking about the same concept—but, promise, no one is going to be asked to solve any quadratic equations.

```
a = 1
nextobj = parent(obj)
temp_word = "the"
```

Global variables are visible throughout the program. They must be defined similarly to properties and attributes as

```
global <global variable name>[ = <initial value>]
```

Local variables, on the other hand, are recognized only within the routine in which they are defined. They are defined using

```
local <local variable name>[ = <initial value>]
```

Global variables must of course have a unique name, different from that of any other data object; local variables, on the other hand, may share the names of local variables in other routines.

In either case, global or local, the default initial value is 0 if no other value is given. For example,

```
global time_of_day = 1100
```

is equal to 1100 when the program is run, and is visible at any point in the program, by any object or routine. On the other hand, the variables

```
local a, max = 100, t
```

are visible only within the block of code where they are defined, and are initialized to 0, 100, and 0, respectively, each time that section of code (be it a routine, property routine, event, etc.) is run.

The compiler defines a set of engine globals: global variables that are referenced directly by the engine, but which may otherwise be treated like any other global variables. These are:

`object`	direct object of an action
`xobject`	indirect object
`self`	self-referential object
`words`	total number of words in command
`player`	the player object
`actor`	the player, or character obj. (for scripts)
`verbroutine`	specified by the command
`location`	location of the `player` object

`endflag`	if not false (0), run `EndGame` routine
`prompt`	for input; default is “>“
`objects`	the total number of objects
`system_status`	after certain operations

The `object`, `xobject`, and `verbroutine` globals are set up by the engine depending on what command is entered by the player. The `self` global is undefined except when an object is being referenced (as in a property routine or event). In that case, it is set to the number of that object. The `player` variable holds the number of the object that the player is controlling; the `endflag` variable must be 0 unless something has occurred to end the game; and the `prompt` variable represents the dictionary word appearing at the start of an input line (which most programs set to “>” by convention).

The `objects` variable can be set by the program, but to no useful effect. The engine will reset it to the “real” value whenever referenced. (All object numbers range from 0 to the value of `objects`.) The `system_status` variable may be read (after a resource operation or a `system` call; see the relevant sections for an explanation of these functions) to check for an error value. See the section on *“Resources”* for possible return values.

Note: Setting `endflag` to a non-zero value forces an *immediate* break from the game loop. Statements following the `endflag` assignment even in the same function are not executed; control is passed directly to the engine, which calls the `EndGame` routine.

IV.b. Constants

Constants are simply labels that represent a non-modifiable value.

```
constant FIRST_NAME "John"
constant LAST_NAME "Smith"
```

(`Note the` lack of an ‘=’ sign between, for example, `FIRST_NAME` and “John”.)

```
print LAST_NAME; ", "; FIRST_NAME
```

results in:

Smith, John

Constants can, like any other Hugo data type, be integers, dictionary entries, object numbers, etc.

It is not absolutely necessary that a constant be given a definite value if the constant is to be used as some sort of flag or marker, etc. Therefore,

```
constant THIS_RESULT
constant THAT_RESULT
```

will have unique values from each other, as well as from any other constant defined without a specific value.

Sometimes it may be useful to enumerate a series of constants in sequence. Instead of defining them all individually, it is possible to use:

```
enumerate start = 1
{
     MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY
}
```

giving:

```
MONDAY = 1, TUESDAY = 2, WEDNESDAY = 3,
THURSDAY = 4, FRIDAY = 5
```

The start value is optional. If omitted, it is 0. Also, it is possible to change the current value at any point (therefore also affecting all following values).

```
enumerate
{
     A, B, C = 5, D, E
}
```

giving:

```
A = 0, B = 1, C = 5, D = 6, E = 7.
```

Finally, it is possible to alter the step value of the enumeration using the `step` keyword followed by `+n`, `-n`, `*n`, or `/n`, where `n` is a constant integer value. To start with 1 and count by multiples of two:

```
enumerate step *2
{
     A = 1, B, C, D
}
```

giving:

```
A = 1, B = 2, C = 4, D = 8.
```

Enumeration of global variables is also possible, using the globals specifier, as in:

```
enumerate globals
{
     <global1>, <global2>,...
}
```

Otherwise the specifier "constants" (as opposed to "globals") is implied as the default.

IV.c. Printing Text

Text can be printed—that is, output to the screen during running of a Hugo program—using two different methods. The first is the basic print command, the simplest form of which is

```
print "<string>"
```

where <string> consists of a series of alphanumeric characters and punctuation.

The backslash character ('\') is handled specially. It modifies how the character following it in a string is treated.[22]

\"	inserts quotation marks
\\	insert a literal backslash character
_	insert a forced space, overriding left-justification for the rest of the string
\n	insert a forced newline

As usual, a single '\' at the end of a line signals that the line continues with the following line.

Examples:

[22] These formatting combinations are valid for printing only; they are not treated as literal characters, as in, for example, expressions involving dictionary entries. Note also that (unlike in languages such as C) formatting sequences such as "\n" are treated as two characters in a string.

```
print "\"Hello!\""
```

```
"Hello!"
```

```
print "Print a...\n...newline"
```

```
Print a...
...newline
```

```
print "One\\two\\three"
```

```
One\two\three
```

```
print "      Left-justified"
print "\_    Not left-justified"
```

```
Left-justified
     Not left-justified
```

```
print "This is a \
  single line."
```

```
This is a single line.
```

(Although

```
print "This is a
     single line."
```

will produce the same result, since the line break occurs within quotation marks.)

After each of the above print commands, a newline is printed. To avoid this, append a semicolon (';') to the end of the `print` statement.

```
print "This is a ";
print "single line."
```

```
This is a single line.
```

Print statements may also contain data types, or a combination of data types and strings. The command

```
print "The "; object.name; " is closed."
```

will print the word located at the dictionary address specified by `object.name`, so that if `object.name` points to the word "box", the resulting output would be:

```
The box is closed.
```

To capitalize the first letter of the specified word, use the `capital` modifier.

```
print "The "; capital object.name; " is closed."
```

```
The Box is closed.
```

To print the data type as a value instead of referencing the dictionary, use the `number` modifier. For example, if the variable `time` holds the value 5,

```
print "There are "; number time; " seconds remaining."
```

```
There are 5 seconds remaining.
```

If `number` were not used, the engine would try to find a word at the dictionary address 5, and the result will likely be garbage.

Mainly for debugging purposes, the modifier `hex` prints the data type as a hexadecimal number instead of a decimal one. If the variable `val` equals 127,

```
print number val; " is "; hex val; " in hexadecimal."
```

```
127 is 7F in hexadecimal.
```

The second way to print text is from the text bank, from which—if memory is in short supply—sections are loaded from disk only when they are needed by the program. This method is provided so that lengthy blocks of text—such as description and narration—do not take up valuable space if memory is limited. The command consists simply of a quoted string without any preceding statement.

```
"This string would be written to disk."
```

```
This string would be written to disk.
```

or

```
"So would this one ";
"and this one."
```

```
So would this one and this one.
```

Notice that a semicolon at the end of the statement still overrides the newline. The in-string formatting combinations are still usable with these print statements, but since each statement is a single line, data types and other modifiers may not be compounded. Because of that,

```
"\"Hello,\"" he said."
```

will write

```
“Hello,” he said.
```

to the .HEX file text bank, but

```
"There are "; number time_left; " seconds remaining."
```

is illegal.

The color of text may be changed using the `color` command (also valid with the U.K. spelling `colour`). The format is

```
color <foreground>[, <background>[, <input color>]]
```

where the background color is not necessary. If no background color is specified, the current one is assumed). The input color is also not necessary—this refers to the color of player input and, if not given, is the same as the foreground color.

The standard color set with corresponding values and constant labels (defined in **hugolib.h**) is:

COLOR	VALUE	LABEL
Black	0	BLACK
Blue	1	BLUE
Green	2	GREEN
Cyan	3	CYAN
Red	4	RED
Magenta	5	MAGENTA
Brown	6	BROWN
White	7	WHITE
Dark gray	8	DARK_GRAY
Light blue	9	LIGHT_BLUE
Light green	10	LIGHT_GREEN
Light cyan	11	LIGHT_CYAN

Light red	12	`LIGHT_RED`
Light magenta	13	`LIGHT_MAGENTA`
Yellow	14	`YELLOW`
Bright white	15	`BRIGHT_WHITE`
Default foreground	16	`DEF_FOREGROUND`
Default background	17	`DEF_BACKGROUND`
Default statusline (fore)	18	`DEF_SL_FOREGROUND`
Default statusline (back)	19	`DEF_SL_BACKGROUND`
Match foreground	20	`MATCH_FOREGROUND`

(Since the labels are defined in **hugolib.h,** when using the library, it is never necessary to refer to a color by its numerical value.)

It is expected that, regardless of the system, any color will print visibly on any other color. Video technology and shortcomings of the visible light spectrum conspire to foil this plan, however, it is suggested for practicality that white (and less frequently bright while) be used for most text-printing. Blue and black are fairly standard background colors for light-colored (such as white) text—this is a common combination for default text (as is dark text, such as black, on a white background). A game author can use the DEF_FOREGROUND, DEF_BACKGROUND, DEF_SL_FOREGROUND, and DEF_SL_BACKGROUND colors (as is done in **sample.hug** and is the default in **shell.hug**) since this uses the colors supplied by the Hugo Engine, allowing the user to change colors to his or her liking if the port supports that capability.

Magenta printing on a cyan background is accomplished by

```
color MAGENTA, CYAN
```

or

```
color 5, 3                ! if not using HUGOLIB.H
```

A current line can be filled—with blank spaces in the current color—to a specified column (essentially a tab stop) using the "print to..." structure as follows:

```
print "Time:"; to 40; "Date:"
```

where the value following to does not exceed the maximum line length in the engine global linelength.

The resulting output will be something like:

```
Time:                              Date:
```

Text can be specifically located using the `locate` command via

```
locate <column>, <row>
```

where

```
locate 1, 1
```

places text output at the top left corner of the current text window. Neither `<column>` nor `<row>` may exceed the current window boundaries—the engine will automatically constrain them as necessary.

IV.d. More Formatting Sequences

As listed above, the following are valid printing sequences that may be embedded in printed strings:

`\"`	quotation marks
`\\`	a literal backslash character
`_`	a forced space, overriding left-justification for the rest of the string
`\n`	a newline

The next set of formatting sequences control the appearance of printed text by turning on and off boldface, italic, proportional, and underlined printing. Not all computers and operating systems are able to provide all types of printed output; however, the engine can be relied upon to properly process any formatting—i.e., proportionally printed text will still look fine even on a system that has only a fixed-width font, such as a Unix text terminal or DOS output (although, of course, it won't be proportionally spaced).

`\B`	boldface on
`\b`	boldface off
`\I`	italics on
`\i`	italics off
`\P`	proportional printing on
`\p`	proportional printing off
`\U`	underlining on
`\u`	underlining off

A statement like the following:

```
"A \Bbold string with some \Iitalics\i and
\Uunderline\b thrown in.\u"
```

will result in output like:

A **bold string with some *italics* and <u>underline</u>** <u>thrown in.</u>

Print style can also be changed using the `Font` routine in **hugolib.h**, so that in

```
Font(<font change code>)
```

the `<font change code>` can be one or more of:

```
BOLD_ON             BOLD_OFF

ITALICS_ON          ITALICS_OFF

UNDERLINE_ON        UNDERLINE_OFF
```

and can subsequently be used alone or in combination such as:

```
Font(BOLD_ON | ITALICS_ON | PROP_OFF)
```

It's preferable to rely on the `Font` function and the various font constants instead of embedding multiple font-change formatting sequences because if for no other reason than it being clearer to understand when reading the source code.

Special characters can also be printed via formatting sequences. Note that these characters are contained in the Latin-1 character set; if a particular system is incapable of displaying it, it will display the normal-ASCII equivalent. (The following examples, appearing in parentheses, may not display properly on all computers and printers.)

\\\`	accent grave	followed by a letter e.g. "\\\`a" will print an 'a' with an accent grave (à)
\\'	accent acute	followed by a letter e.g. "\\'E" will print an 'E' with an accent acute (É)
\\~	tilde	followed by a letter

e.g. "\~n" will print an 'n' with a tilde (ñ)

`\^` circumflex — followed by a letter
e.g. "\^i" will print an 'i' with a circumflex (î)

`\:` umlaut — followed by a letter
e.g. "\:u" will print a 'u' with an umlaut (ü)

`\,` cedilla — followed by c or C
e.g. "\,c" will print a 'c' with a cedilla (ç)

\< or \>	Spanish quotation marks	(« »)
`\!`	upside-down exclamation point	(¡)
`\?`	upside-down question mark	(¿)
`\ae`	ae ligature	(æ)
`\AE`	AE ligature	(Æ)
`\c`	cents symbol	(¢)
`\L`	British pound	(£)
`\Y`	Japanese Yen	(¥)

`\#xxx` any ASCII or Latin-1 character where xxx represents the three-digit ASCII number (or Latin-1 code) of the character to be printed, e.g. "`\#065`" will print an 'A' (ASCII 65) (Care should be taken when using codes other than those for which special character support explicitly exists, as not all systems or fonts may display all desired non-ASCII characters.)

Note: It is possible to embed Latin-1 characters directly into printed text in source code using a text editor that allows it—but ensure that the non-ASCII characters are indeed Latin-1. Using non-Latin-1 fonts (such as Mac-encoded fonts or other encodings) will result in the wrong character(s) being printed on various platforms. Also note that platforms which *cannot* display Latin-1 characters (including some Unix-based terminal displays, DOS windows, etc.) may not have proper Latin-1-to-ASCII translation in order to *decode* Latin-1 characters embedded directly in printed text. For this reason, or if you're not positive whether your font encoding is Latin-1, stick to using the special-character sequences described above, which are guaranteed to work properly across platforms.

IV.e. Operators and Assignments

Hugo allows use of all standard mathematical operators:

`*`	multiplication
`/`	integer division

which take precedence[23] over:

`+`	addition
`-`	subtraction

Comparisons are also valid as operators, returning Boolean true or false (1 or 0) so that

```
2 + (n = 1)
5 - (n > 1)
```

evaluate respectively to 3 and 5 if `n` is 1, and 2 and 4 if `n` is 2 or greater. Valid relational operators are

`=`	equal to
`~=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

Logical operators (`and`, `or`, and `not`) are also allowed.

```
(x and y) or (a and b)
(j + 5) and not ObjectisLight(k)
```

Using `and` results in true (1) if both values are non-zero. Using `or` results in true if either is non-zero; `not` results in true only if the following value is zero.

```
1 and 1 = 1
1 and 0 = 0
5 and 3 = 1
0 and 9 = 0
0 and 169 and 1 = 0
```

[23] Hugo follows standard order of operations for operator precedence.

```
1 and 12 and 1233 = 1

1 or 1 = 1
35 or 0 = 1
0 or 0 = 0

not 0 = 1
not 1 = 0
not 8 = 0
not (8 and 0) = 1

1 and 7 or (14 and not 0) = 1
(0 or not 1) and 3 = 0
```

Additionally, bitwise operators are provided:

```
1 & 1 = 1          (Bitwise and)
1 & 0 = 0

1 | 0 = 1          (Bitwise or)
1 | 1 = 1

~0 = -1            (Bitwise not/inverse)
```

(As mentioned previously, a detailed explanation of bitwise operations is a little beyond the scope of this manual; programmers may occasionally use the '`|`' operator to combine bitmask-type parameters for certain library functions such as fonts and list-formats, but only advanced users should have to worry about employing bitwise operators to any great extent in practical programming.)

Any Hugo data type can appear in an expression, including routines, attribute tests, properties, constants, and variables. Standard mathematical rules for order of significance in evaluating an expression apply, so that parenthetical sub-expressions are evaluated first, followed by multiplication and division, followed by addition and subtraction.

Some sample combinations are:

```
10 + object.size          ! integer constant and
                          !   property

object is openable + 1    ! attribute test and constant

FindLight(location) + a   ! return value and variable
```

```
1 and object is light    ! constant, logical test,
                         !    and attribute
```

Expressions can be evaluated and assigned to either a variable, a property, or an array element.

```
<variable> = <expression>

<object>.<property> [#<element>] = <expression>

<array>[<element>] = <expression>
```

IV.f. Efficient Operators

Something like

```
number_of_items = number_of_items + 1
if number_of_items > 10
{
     print "Too many items!"
}
```

can be coded more simply as

```
if ++number_of_items > 10
{
     print "Too many items!"
}
```

The '++' operator increases the following variable by one before returning the value of the variable. Similarly, '--' can precede a variable to decrease the value by one before returning it. Since these operators act before the value is returned, they are called "pre-increment" and "pre-decrement".

If '++' or '--' comes *after* a variable, the value of the variable is returned and then the value is increased or decreased, respectively. In this usage, the operators are called "post-increment" and "post-decrement".

For example,

```
while ++i < 5       ! pre-increment
{
     print number i; " ";
}
```

will output:

```
1 2 3 4
```

But

```
while i++ < 5        ! post-increment
{
     print number i; " ";
}
```

will output:

```
1 2 3 4 5
```

Since in the second example, the variable is increased before getting the value, while in the second example, it is increased after checking it.

It is also possible to use the operators '`+=`', '`-=`', '`*=`', '`/=`', '`&=`', and '`|=`'. These can also be used to modify a variable at the same time its value is being checked. All of these, however, operate before the value in question is returned.

```
x = 5
y = 10
print "x = "; number (x*=y); ", y = "; number y
```

Result:

```
x = 50, y = 10
```

When the compiler is processing any of the above lines, the efficient operator takes precedence over a normal (i.e., single-character) operator. For example,

```
x = y + ++z
```

is actually compiled as

```
x = y++ + z
```

since the '++' is parsed first. To properly code this line with a pre-increment on the z variable instead of a post-increment on y, use parentheses to order the various operators:

```
x = y + (++z)
```

IV.g. Arrays And Strings

Prior to this point, little has been said about arrays. Arrays are sets of values that share a common name, and where the elements are referenced by number. Arrays are defined by

```
array <arrayname> [<array size>]
```

where `<array size>` must be a numerical constant.

An array definition reserves a block of memory of `<array size>`[24], so that, for example,

```
array test_array[10]
```

reserves ten possible storage elements for the array.

Keep in mind that `<array size>` determines the size of the array, not the maximum element number. Elements begin counting at 0, so that `test_array`, with 10 elements, has members numbered from 0 to 9. Trying to access `test_array[10]` or higher will return a zero value (and, if running in the debugger, cause a debugger warning). Trying to assign it by mistake will have no effect.

To prevent such out-of-bounds array reading/writing, an array's length may be read via:

```
array[]
```

where no element number is specified. Using the above example,

```
print number test_array[]
```

would result in "10".

Array elements can be assigned more than one at a time, as in

```
<arrayname> = <element1>, <element2>, ...
```

where `<element1>` and `<element2>` can be expressions or single values.

Elements need not be all of the same type, either, so that

```
test_array[0] = (10+5), "Hello!", FindLight(location)
```

[24] Measured in 16-bit words, or 2 bytes per element.

is perfectly legal (although perhaps not perfectly useful). More common is a usage like

```
names[0] = "Ned", "Sue", "Bob", "Maria"
```

or

```
test_array[2] = 5, 4, 3, 2, 1
```

The array can then be accessed by

```
print names[0]; " and "; names[3]
```

```
Ned and Maria
```

or

```
b = test_array[3] + test_array[5]
```

which would set the variable `b` to 4 + 2, or 6.

Because array space is statically allocated by the compiler, all arrays must be declared at the global level. Local arrays are illegal, as are entire arrays passed as arguments[25]. However, single elements of arrays are valid arguments.

It is, however, possible to pass an array address as an argument, and the routine can then access the elements of the array using the `array` modifier. For example, if items is an array containing:

```
items[0] = "apples"
items[1] = "oranges"
items[2] = "socks"
```

The following:

```
routine Test(v)
{
	print array v[2]
}
```

can be called using

```
Test(items)
```

[25] "Arguments" are simply parameters passed to a routine at calling time. See *V.a Routines*.

to produce the output

```
socks
```

even though `v` is an argument (i.e., local variable), and technically not an array. The line

```
print array v[2]
```

tells the engine to treat `v` as an array address, so that we can follow it with `[<element number>]`.

Arrays also allow a Hugo programmer to implement what are known as *string arrays*, which are textual strings, somewhat similar but not identical to dictionary entries. Most significantly, since they are arrays, string arrays may be altered at runtime by a program (unlike dictionary entries, which are hard-coded into the program's dictionary). A string array is an array containing a series of character values, terminated by a zero value.

If the array `apple_array` holds the string array "apple", the actual elements of `apple_array` look like:

```
apple_array[0] = 'a'
apple_array[1] = 'p'
apple_array[2] = 'p'
apple_array[3] = 'l'
apple_array[4] = 'e'
apple_array[5] = 0
```

Hugo provides a handy way to store a dictionary entry in an array as a series of characters using the `string` built-in function:

```
string(<array address>, <dict. entry>, <max. length>)
```

For example,

```
string(a, word[1], 10)
```

will store up to 10 characters from `word[1]` into the array `a`.

Note: It is expected in the preceding example that `a` would have at least 11 elements, since `string` expects to store a terminating 0 after the string itself.

It's not necessary to look at the return value from string, but it can be useful, since it lets us know how many characters were written to the string. For example,

```
x = string(a, "microscopic", 10)
```

will store up to 10 characters of "microscopic" in the array a, and return the length of the stored string to the variable x.[26]

The Hugo Library defines the functions StringCopy, StringEqual, StringLength, and StringPrint, which are extremely useful when dealing with string arrays.

StringCopy copies one string array to another array.

```
StringCopy(<new array>, <old array>[, <length>])
```

For example,

```
StringCopy(a, b)
```

copies the contents of b to a, while

```
StringCopy(a, b, 5)
```

copies only up to 5 characters of b to a.

```
x = StringEqual(<string1>, <string2>)
x = StringCompare(<string1>, <string2>)
```

StringEqual returns true only if the two specified string arrays are identical. StringCompare returns 1 if <string1> is lexically greater than <string2>, -1 if <string1> is lexically less than <string2>, and 0 if the two strings are identical.

StringLength returns the length of a string array, as in:

```
len = StringLength(a)
```

and StringPrint prints a string array (or part of it).

```
StringPrint(<array address>[, <start>, <end>])
```

---

[26] (The built-in engine variables 'parse$' and 'serial$' may be used in place of the dictionary entry address; see *VII.b The Parser* for a description.)

For example, if the array a contains "presto",

```
StringPrint(a)
```

will print "presto", but

```
StringPrint(a, 1, 4)
```

will print "res". (The <start> parameter in the first example defaults to 0, not 1—remember that the first numbered element in an array is 0.)

An interesting side-effect of being able to pass array addresses as arguments is that it is possible to "cheat" the address, so that, for example,

```
StringCopy(a, b+2)
```

will copy b to a, beginning with the third letter of b (since the first letter of b is b[0]).

It should also be kept in mind that string arrays and dictionary entries are two entirely separate animals, and that comparing them directly is using StringCompare is not possible. That is, while a dictionary entry is a simple value representing an address, a string array is a series of values each representing a character in the string.

The library provides the following to overcome this:

```
StringDictCompare(<array>, <dict. entry>)
```

which returns the same values (1, -1, 0) as StringCompare, depending on whether the string array is lexically greater than, less than, or equal to the dictionary entry.

There is also a complement to string: the dict built-in function, that dynamically creates a new dictionary entry at runtime. Its syntax is:

```
x = dict(<array>, <maxlen>)
x = dict(parse$, <maxlen>)
```

where the contents of <array> or parse$ are written into the dictionary, to a maximum of <maxlen> characters, and the address of the new word is returned.

However, since this requires extending the actual length of the dictionary table in the game file, it is necessary to provide for this during compilation. Inserting

```
$MAXDICTEXTEND=<number>
```

at the start of the source file will write a buffer of `<number>` empty bytes at the end of the dictionary. (`MAXDICTEXTEND` is, by default, 0.)

Dynamic dictionary extension is used primarily in situations where the player may be able to, for example, name an object, then refer to that object by the new name, or whenever the game needs to introduce new words into the dictionary that are not known at compile-time. However, a guideline for programmers is that there should be a limit to how many new words the program or player can cause to be created, so that the total length of the new entries never exceeds `<number>`, keeping in mind that the length of an entry is the number of characters plus one (the byte representing the actual length). That is, the word "test" requires 5 bytes.)

IV.h. Conditional Expressions and Program Flow

Program flow can be controlled using a variety of constructions, each of which is built around an expression that evaluates to false (zero) or non-false (non-zero).

The most basic of these is the `if` statement.

```
if <expression>
     {...conditional code block...}
```

The enclosing braces are not necessary if the code block is a single line. Note also that the conditional block may begin (and even end) on the same line as the `if` statement provided that braces are used.

```
if <expression>
     ...single line...

if <expression> {...conditional code block...}
```

If braces are not used for a single line, the compiler automatically inserts them, although special care must be taken when constructing a block of code nesting several single-line conditionals. While

```
if <expression1>
     if <expression2>
          ...conditional code block...
```

may be properly interpreted, other constructions (particularly those involving some of the more complex program-flow concepts we're about to get into) may

not be. Therefore, it's always best to be as clear as possible about your intent, more along the lines of:

```
if <expression1>
{
     if <expression2>
               ...conditional code block...
}
```

More elaborate uses of `if` involve the use of `elseif` and `else`.

```
if <expression1>
     ...first conditional code block...
elseif <expression2>
     ...second conditional code block...
elseif <expression3>
     ...third conditional code block...
...
else
     ...default code block...
```

In this case, the engine evaluates each expression until it finds one that is true, and then executes it. Control then passes to the next non-`if/elseif/else` statement following the conditional construction. If no true expression is found, the default code block is executed. If, for example, `<expression1>` evaluates to a non-false value, then none of the following expressions are tested.

Of course, all three (`if`, `elseif`, and `else`) need not be used every time, and simple `if-elseif` and `if-else` combinations are perfectly valid.

In certain cases, the `if` statement may not lend itself perfectly to clarity, and the `select-case` construction may be more appropriate. The general form is:

```
select <var>
     case <value1>[, <value2>, ...]
          ...first conditional code block...
     case <value3>[, <value4>, ...]
          ...second conditional code block...
     ...
     case else
          ..default code block...[27]
```

[27] C programmers are used to cases that "fall through" to the next case unless explicitly told not to do so; such is not the case with Hugo.

In this case, the evaluation is essentially

```
if <var> = <value1> [or <var> = <value2> ...]
```

There is no limit on the number of values (separated by commas) that can appear on a line following `case`[28]. The same rules for bracing multiple-line code blocks apply as with `if` (as well as for every other type of conditional block).

Basic loops may be coded using `while` and `do-while`.

```
while <expression>
     ...conditional code block...

do
     ...conditional code block...
while <expression>
```

Each of these executes the conditional code block as long as `<expression>` holds true. It is assumed that the code block somehow alters `<expression>` so that at some point it will become false; otherwise the loop will execute endlessly.

```
while x <= 10
{
     x = x + 1
     print "x is "; number x
}

do
{
     x = x + 1
     print "x is "; number x
}
while x <= 10
```

The only difference between the two is that if `<expression>` is false at the outset, the `while` code block will never run. The `do-while` code block will run at least once even if `<expression>` is false at the outset.

It is also important to recognize—with `while` or `do-while` loops—that the expression is tested each time the loop executes. The most important side effect of this is that if you're doing something in the expression that has some

---

[28] Okay, this isn't quite true. While there isn't an *explicit* limit, if you create a single '`case`' line that runs on forever and ever, eventually you'll reach the point where, for buffer reasons, the compiler isn't able to compile it, and it will complain with an appropriate error.

effect—whether printing something, calling a function, or modifying some other value—this will happen *every time* the expression is evaluated.

The most complex loop construction uses the for statement:

```
for (<assignment>; <expression>; <modifier>)
     ...conditional code block...
```

For example:

```
for (i=1; i<=15; i=i+1)
     print "i is equal to: "; number i
```

First, the engine executes the assignment setting "i = 1". Next, it checks to see if the expression holds true (if i is less than or equal to 15). If it does, it executes the print statement and the modifying assignment that increments i. It continues the loop until the expression tests false.

Not all elements of the for construction are necessary. For example, the assignment may be omitted, as in

```
for (; i<=15; i=i+1)
```

and the engine will simply use the existing value of i, whatever it was before this point. With

```
for (i=1;;i=i+1)
```

the loop will execute endlessly, unless some other means of exit is provided.

The modifying expression does not have to be an arithmetic expression as shown above. It may be a routine that modifies a global variable, for example, which is then tested by the for loop.

A second form of a for loop is:

```
for <var> in <object>
     ...conditional code block...
```

which loops through all the children of <object> (if any), setting the variable <var> to the object number of each child in sequence, so that

```
for i in mysuitcase
     print i.name
```

will print the names of each object in the mysuitcase object.

Hugo also supports `jump` commands and labels. A label is simply a user-specified token preceded by a colon (':') at the beginning of a line. The label name must be a unique token in the program.[29]

```
     print "We're about to make a jump."
     jump NextLine
     print "This will never get printed."
:NextLine
     print "But this will."
```

One final concept is important in program flow, and that is `break`. At any point during a loop, it may be necessary to exit immediately (and probably prematurely). The `break` statement passes control to the statement immediately following the current loop. In the example:

```
do
{
     while <expression2>
     {
          ...
          if <expression3>
               break
          ...
     }
     ...
}
while <expression1>
```

the `break` causes the immediately running `while <expression2>` loop to terminate, even if `<expression2>` is true. However, the external `do-while <expression1>` loop continues to run.

It has been previously stated that lines ending in `and` or `or` are continued onto the next line in the case of long conditional expressions. A second useful provision is the ability to use a comma to separate options within a conditional expression. As a result,

```
if word[1] = "one", "two", "three"

while object is open, not locked
```

[29] The `jump` keyword is more or less equivalent to `goto` in other languages. The reason it's different in Hugo is mainly to encourage the use of the proper alternatives (i.e., `for` and `while` or `do-while` loops) in keeping with proper programming practices. And, in the end, less `jump`s and labels make for far more readable code.

```
if box not in livingroom, garage

if a ~= 1, 2, 3
```

are interpreted as:

```
if word[1]="one" or word[1]="two" or word[1]="three"

while object is open and object is not locked

if box not in livingroom and box not in garage

if a ~= 1 and a ~= 1 and a ~= 3
```

respectively.

Note that with an '=' or `in` comparison, a comma results in an `or` comparison. With '~=' or an attribute comparison, the result is an `and` comparison. The compiler looks after this translation for you.

IV.i. What Should I Be Able To Do Now?

Example: Mixing Text Styles

```
! Sample to print various typefaces/colors:

#include "hugolib.h"

routine main
{
     print "Text may be printed in \Bboldface\b,
          \Iitalics\i, \Uunderlined\u, or
          \Pproportional\p typefaces."

     color RED                ! or color 4
     print "\nGet ready.  ";
     color YELLOW             ! color 14
     print "Get set.  ";
     color GREEN              ! color 2
     print "Go!"
}
```

The output will be:

```
Text may be printed in boldface, italics, underlined,
or proportional typefaces.

Get ready.  Get set.  Go!
```

with "boldface", "italics", "underlined", and "proportional" printed in their respective typefaces. "Get ready", "Get set", and "Go!" will all appear on the same line in three different colors.

Note that not all computers will be able to print all typefaces. The basic Unix and MS-DOS ports, for example, use color changes instead of actual typeface changes, and do not support proportional printing.

Example: Managing Strings

```
#include "hugolib.h"

routine main
```

```
{
     StringTests
     return
}

array s1[32]
array s2[10]
array s3[10]

routine StringTests
{
     local a, len

     a = "This is a sample string."
     len = string(s1, a, 31)
     string(s2, "Apple", 9)
     string(s3, "Tomato", 9)

     print "a = \""; a; "\""
     print "(Dictionary address:  "; number a; ")"
     print "s1 contains \""; StringPrint(s1); "\""
     print "(Array address:  "; number s1;
     print ", length = "; number len; ")"
     print "s2 is \""; StringPrint(s2);
     print "\", s3 is \""; StringPrint(s3); "\""

     "\nStringCompare(s1, s2) = ";
     print number StringCompare(s1, s2)
     "StringCompare(s1, s3) = ";
     print number StringCompare(s1, s3)
}
```

The output will be:

```
a = "This is a sample string."
(Dictionary address = 887)
s1 contains "This is a sample string."
(Array address = 1625, length = 24)
s2 is "Apple", s3 is "Tomato"

StringCompare(s1, s2) = 1
StringCompare(s1, s3) = -1
```

As is evident above, a dictionary entry does not need to be a single word; any piece of text which is referred to by the program as a value gets entered into the dictionary table.

The argument 31 in the first call to the `string` function allows up to 31 characters from a to be copied to `s1`, but since the length of `a` is only 24 characters, only 25 values (including the terminating 0) get copied, and the string length of `s1` is returned in len.

Since "A(pple)" is lexically less than "T(his...)", comparing the two returns -1. As "To(mato)" is lexically greater than "Th(is...)", `StringCompare` returns 1.

V. ROUTINES AND EVENTS

V.a. Routines

Routines are blocks of code that may be called at any point in a program. A routine may or may not return a value, and it may or may not require a list of *parameters* or *arguments*. (Some different uses of routines have already been encountered in previous examples, but here is the formal explication.) Routines are also sometimes referred to *subroutines* or *functions*, the latter particularly when they're returning a value, i.e., performing some function and reporting the result.

A routine is defined as:

```
routine <routinename> [(<arg1>, <arg2>, ...)]
{
     ...
}
```

once again ensuring that the opening brace ('{') comes on a new line following the `routine` specifier.

Note: To substitute a new routine for an existing one with the same name (such as in a library file), define the new one using `replace` instead of `routine`.

```
replace <routinename> [(<arg1>, <arg2>, ...)]
```

An example of a simple routine definition is:

```
routine TestRoutine(obj)
{
     print "The "; obj.name; " has a size of ";
     print obj.size; "."
     return obj.size
}
```

Where `TestRoutine` takes a single value as an argument, assigns it to a local variable `obj`, executes a simple printing sequence, and returns the property value: `obj.size`.

The `return` keyword exits the current routine, and returns a value if specified. Both

```
return
```

and

```
return <expression>
```

are valid. If no expression is given, the routine returns 0. If no `return` statement at all is encountered, the routine continues until the closing brace ('`}`'), then returns 0.[30]

Once defined, `TestRoutine` can be called several ways:

```
TestRoutine(mysuitcase)
```

will (assuming the `mysuitcase` object as been defined as previously illustrated) print

```
"The big green suitcase has a size of 25."
```

The return value will be ignored. On the other hand,

```
x = TestRoutine(mysuitcase)
```

will print the same output, but will assign the return value of `TestRoutine` to the variable x.

Now, unlike C and similar languages, Hugo does not require that routines follow a strict prototype. Therefore, both

```
TestRoutine
```

and

```
TestRoutine(mysuitcase, 5)
```

---

[30] Routines return 0 by default, with the exception of *property routines*, which we'll discuss shortly and which return true (or 1) by default.

are valid calls for the above routine. In the first case, the argument `obj` defaults to 0, since no value is passed. (The parentheses are not necessary if no arguments are passed.) In the second case, the value 5 is passed to `TestRoutine`, but ignored.

Arguments are always passed by value, not by reference or address. A local variable in one routine can never be altered by another routine. What this means is that, for example, in the following routines:

```
routine TestRoutine
{
     local a

     a = 5
     Double(a)
     print number a
}

routine Double(a)
{
     a = a * 2
}
```

calling `TestRoutine` would print "5" and not "10" because the local variable `a` in `Double` is only a copy of the variable passed to it as an argument.

These two routines would, on the other hand, print "10":

```
routine TestRoutine
{
     local a

     a = 5
     a = Double(a)
     print number a
}

routine Double(a)
{
     return a * 2
}
```

The local `a` in `TestRoutine` is reassigned with the return value from the `Double` routine.

An interesting side-effect of a zero (0) return value can be seen using the `print` command[31]. Consider the `The` routine in **`hugolib.h`**, which prints an object's definite article (i.e., "the", if appropriate), followed by the object's `name` property.

```
print "You open "; The(object); "."
```

might result in

```
You open the suitcase.
```

Note that the above `print` command itself really only prints

```
"You open "
```

and

```
"."
```

It is the `The` routine that prints

```
the suitcase
```

Since `The` returns 0 (the empty string, or ""), the `print` command is actually displaying

```
"You open ", "", and "."
```

where the empty string ("") is preceded on the output line by `The`'s printing of "the " (notice the trailing space) and the object name.

V.b. Property Routines

Property routines are slightly more complex than the simple routines described so far, but follow the same basic rules. Normally, a property routine runs when the program attempts to get the value of a property that contains a routine.

That is, instead of having the property value:

```
size 10
```

[31] Remember here that both zero (0) and the empty string "" are equal in value.

an object may contain the property:

```
size
{
     return some_value + 5
}
```

Trying to read `object.size` in either case will return an integer value, although in the second case it is calculated by a routine.

Note: While normal routines return false (or 0) by default, property routines return true (or 1) by default.

Here's another example. Normally, if `<object>` is the current room object, then `<object>.n_to` would contain the object number of the room to the north (if there is one). The library checks `<object>.n_to` to see if a value exists for it; if none does, the move is invalid.

Consider this:

```
n_to office
```

and

```
n_to
{
     "The office door is locked."
}
```

or

```
n_to
{
     "The office door is locked.  ";
     return false
}
```

In the first case, an attempt on the part of the player to move north would result in `parent(player)` being changed to the `office` object. In the second case, a custom invalid-move message would be displayed. In the third case, the custom invalid-move message would be displayed, but then the library would continue as if it had not found a `n_to` property for `<object>`, and it would print the standard invalid-move message (without a newline, thanks to the semicolon):

```
"The office door is locked.  You can't go that way."
```

(For those wondering why the true (i.e., equal to 1) return value in the second case doesn't prompt a move to object number 1, the library DoGo routine assumes that there will never be a room object numbered one, since there are all manner of system objects that get defined first.)

Property routines may be run directly using the run command:

```
run <object>.<property>
```

If <object> does not have <property>, or if <object>.<property> is not a routine, nothing happens. Otherwise, the property routine executes. Property routines do not take arguments.

Remember that at any point in a program, an existing property may be changed using

```
<object>.<property> = <value>
```

A property routine may be changed using

```
<object>.<property> =
{
     ...the new code for this property routine...
}
```

where the new routine must be enclosed in braces.

It is entirely possible to change what was once a property routine into a simple value, or vice-versa, providing that space for the routine (and the required number of elements) was allowed for in the original object definition. Even if a property routine is to be assigned later in the program, the property itself must still be defined at the outset in the original object definition. A simple

```
<property> 0
```

or

```
<property> {return false}
```

will suffice.

There is, however, one drawback to this reassignment of property values to routines and vice-versa. A property routine is given a "length" of one value, which is the property address. When assigning a value or set of values to a property routine, the engine behaves as if the property was originally defined for

this object with only one word of data, since it has no way of knowing the original length of the property data.

For example, if the original property specification in the object definition was:

```
found_in bedroom, livingroom, garage
```

and at some point the following was executed:

```
found_in = { return basement }
```

then the following would not subsequently work:

```
found_in #3 = attic
```

because the engine now believes `<object>.found_in` to have only one element—a routine address—attached to it.

Finally, keep in mind that whenever calling a property routine, the global variable `self` is normally set to the object number. To avoid this, such as when "borrowing" a property from another object from within a different object, reference the property via

```
<object>..<property>
```

using '`..`' instead of the normal property operator.

V.c. Before And After Routines

The Hugo Compiler predefines two special properties: `before` and `after`. They are unique in that not only are they always routines, but they are much more complex (and versatile) than a standard property routine.

Complex properties like `before` and `after` are defined with

```
property <property name> $complex
```

as in:

```
property before $complex
property after  $complex
```

Here is the syntax for the `before` property:

```
before
{
	<usage1> <verbroutine1>[, <verbroutine2>,...]
	{
		...
	}
	<usage2> <verbroutine3>[, <verbroutine4>,...]
	{
		...
	}
	...
}
```

(The `after` property is the same, substituting `after` for `before`.)

The `<usage>` specifier is a value against which the specified object is matched. Most commonly, it is "`object`", "`xobject`", "`location`", "`actor`", "`parent(object)`", etc. The `<verbroutine>` is the name of a verb routine to which the usage in question applies.

When the Hugo Engine goes to execute a player command, it runs a series of tests on the various elements of the command, such as the object on which the specified verb is to be enacted[32]. Know for now that when a player command is executed, the `before` properties of the object (i.e., the direct object) and xobject (i.e., the indirect object)[33] are checked, then if neither has returned non-false, the appropriate `verbroutine` is run. Afterward, the `after` properties are checked; if neither returns non-false, a default message is normally printed by the verbroutine. In other words, `before` routines typically pre-empt the execution of a verbroutine, and `after` routines typically pre-empt the default response of a verbroutine.

When the `<object>.before` property is checked, with the global `verbroutine` set to one of the specified verbroutines in the before property, and `<usage>` in that instance is "`object`", then the following block of code is executed. If no match is found, `<object>.before` returns false.

Here is an example applied to the `mysuitcase` object created previously:

```
before
{
	object DoEat
	{
		"You can't eat the suitcase!"
	}
```

[32] The actual mechanics are described in *VIII.g Perform*.

[33] In the imperative sentence "Put the book on the shelf", the book is the direct object, and the shelf is the indirect object.

```
}

after
{
     object DoGet
     {
          "With a vigorous effort, you pick up
          the suitcase."
     }
     xobject DoPutIn
     {
          "You put ";
          The(object)
          " into the suitcase."
     }
}
```

When the player tries the command “eat suitcase”, the response printed will be:

```
You can't eat the suitcase!
```

and the normal verbroutine for “eat”, the library’s `DoEat` verbroutine, will not be run. When the player tries to “get the suitcase”, the library’s `DoGet` verbroutine will be run (since no `before` property interrupts it), but instead of the default library response (which is a simple “Taken.”), the game will print:

```
With a vigorous effort, you pick up the suitcase.
```

Finally, when the player tries to put something into the suitcase using, say, “put the book in the suitcase”, the normal `DoPutIn` routine will be run, but the custom response of the suitcase will be printed instead:

```
You put the book into the suitcase.
```

Each of these examples will return true (as property routines do by default), thereby overriding the engine’s default operation[34]. In order to fool the engine into continuing normally, as if no `before` or `after` property has been found, return false from the property routine.

```
after
{
     object DoGet
```

---

[34] See *IX THE GAME LOOP*.

```
        {
                "Fine.  ";
                return false
        }
}
```

will result in:

```
>GET SUITCASE
Fine.  Taken.
```

Since the after routine returns false, and the library's default response for a successful call to `DoGet` is "Taken."

It is important to remember that, unlike other property routines, `before` and `after` routines are also *additive*; i.e., a `before` (or `after`) routine defined in an inherited class or object is not overwritten by a new property routine in the new object. Instead, the definition for the routine is—in essence—added onto. An additive property is defined using the `$additive` qualifier, as in:

```
property <property name> $additive <default value>
```

All previously inherited `before`/`after` subroutines are carried over. However, the processing of a `before`/`after` property begins with the present object, progressing backward through the object's ancestry until a usage/verbroutine match is found; once a match is made, no further preceding class inheritances are processed (unless the property routine in question returns false).

Note: To force a `before` or `after` property routine to apply to *any* verbroutine, do not explicitly specify a verbroutine.

For example:

```
before
{
        xobject
        {
                ...property routine...
        }
}
```

The specified routine will be run whenever the object in question is the xobject of any valid input.

If a non-specific block occurs before any block(s) specifying verbroutines, then the following blocks, if matched, will run as well so long as the block does not return true. If the non-specific block comes after any other blocks, then it will run only if no other object/verbroutine combination is matched.

A drawback of this non-specification is that all verbroutines are matched—both verbs and xverbs[35]. This can be particularly undesirable in the case of location `before`/`after` properties, where you may wish to be circumventing any action the player tries to perform in that location, but where the non-specific response will be triggered even for `save`, `restore`, etc. (i.e.,

To get around this, the library provides a function `AnyVerb`, which takes an object as its argument and returns that object number if the current verbroutine is not within the group of xverbs; otherwise it returns false. Therefore, it can be used via:

```
before
{
     AnyVerb(location)
     {
          ...
     }
}
```

instead of

```
before
{
     location
     {
          ...
     }
}
```

The former will execute the conditional block of code whenever the location global matches the current object and the current verbroutine is not an xverb. The latter (without using `AnyVerb`), will run for verbs and xverbs. (The reason for this, simply put, is that the `location` global always equals the `location` global (of course!). But `AnyVerb(location)` will only equal the `location` global if the verbroutine is not an xverb.)

[35] *Verbs* are actions that the player uses to interact with the physical world of the game. *Xverbs* are "non-action" verbs that generally deal with system functions, such as getting help, saving a game, etc. but don't otherwise affect the state of the game world See *VII.a Grammar Definition*.

V.d. Init And Main

At least two routines are typically part of every Hugo problem: `Init` and `Main`. `Init` is optional but almost always implemented. If it exists, is called once at the start of the program (as well as during a `restart` command). The routine should configure all variables, objects, and arrays needed to set up the game state and begin the game. Here's the `Init` routine from **shell.hug**:

```
routine init
{
```

Start the counter at one turn before 0 turns, since `Main` will increment it to begin the game:

```
        counter = -1
```

Set up the kind of statusline we're going to be displaying, as well as define the default text colors[36]:

```
        STATUSTYPE = 1           ! score/turns
        TEXTCOLOR = DEF_FOREGROUND
        BGCOLOR = DEF_BACKGROUND
        SL_TEXTCOLOR = DEF_SL_FOREGROUND
        SL_BGCOLOR = DEF_SL_BACKGROUND
```

Set the player prompt to the default ">", and set the starting foreground and background colors:

```
        prompt = ">"
        color TEXTCOLOR, BGCOLOR
```

Clear the screen before starting the game, set the font to the default font, and print the game title ("SHELL", in this case) and a subtitle, followed by the BANNER constant:

```
        cls
        Font(BOLD_ON | DEFAULT_FONT)
        "SHELL"
        Font(BOLD_OFF)
        "An Interactive Starting Point\n"
        print BANNER
```

*Set the player to the "`you`" object (from **objlib.h**), and set up the starting location:*

---

[36] All of the capitalized `CONSTANTS` used here are defined in **hugolib.h**.

```
        player = you              ! player initialization
        location = emptyroom
        old_location = location
```

Move the player to the starting location, run the library rules to see if there's light in the location, then describe the starting location and flag it as visited. Also, determine the starting bulk of whatever the player is carrying at the outset (if anything):

```
        move player to location
        FindLight(location)
        DescribePlace(location)
        location is visited
        CalculateHolding(player)
```

Finally, if we've defined `USE_PLURAL_OBJECTS`[37], call the appropriate initialization routine:

```
    #ifset USE_PLURAL_OBJECTS
        InitPluralObjects
    #endif
    }
```

`Main` is called every turn. It should take care of general game management such as moving ahead the counter, as well as running events and scripts[38]. The `Main` routine from **`shell.hug`** is as follows:

```
    routine main
    {
```

The `counter` global gets incremented each turn, and the statusline gets updated:

```
        counter = counter + 1
        PrintStatusLine
```

The `each_turn` property of the current `location` object gets run. The `runevents` statement runs all valid events. The `RunScripts` library routine runs any active scripts:

---

[37] A constant that tells **`objlib.h`** that we're implementing a special class of plural/identical objects for use in the game.

[38] Events and scripts are discussed next.

```
        run location.each_turn
        runevents
        RunScripts
```

Finally, we check to see if the player is currently engaged in conversation with a character (if the `speaking` global is set) and, if so, if the character in question has left the current location:

```
        if parent(speaking)~=location
            speaking = 0
    }
```

V.e. Events

Events are useful for bringing a game to life, so that little quirks, behaviors, and occurrences can be provided for with little difficulty or complexity. Events are also routines, but their special characteristic is that they may be attached to a particular object, and they are run as a group by the `runevents` command. Events are defined as:

```
    event
    {
        ...Event routine...
    }
```

for global events, and

```
    event [in] <object>
    {
        ...Event routine...
    }
```

for events attached to a particular object. (The `in` is optional, but is recommended for legibility.) If an event is attached to an object, it is run only when that object has the same grandparent as the player object, where "grandparent" refers to the last object before 0 (the `nothing` object as defined in **`hugolib.h`**), or can otherwise be determined to be in the player's current location[39].

Note: If the event is not a global event, the `self` global is set to the number of the object to which the event is attached.

[39] That is, by the `FindObject` routine in **`hugolib.h`**, as called by the engine.

V.f. What Should I Be Able To Do Now?

Example: "Borrowing" Property Routines

Consider a situation where a class provides a particular property routine. Normally, that routine is inherited by all objects defined using that class. But there may arise a situation where one of those objects must have a variation or expansion on the original routine.

```
class food
{
     bites_left 5
     eating
     {
          self.bites_left = self.bites_left - 1
          if self.bites_left = 0
               remove self         ! all gone
     }
}

food health_food
{
     eating
     {
          actor.health = actor.health + 1
          run food..eating
     }
}
```

(Assuming that `bites_left`, `eating`, and `health` are defined as properties, with `eating` being called whenever a `food` object is eaten.)

In this case, it would be inconvenient to have to retype the entire `food.eating` routine for the `health_food` object just because the latter must also increase `actor.health`. Using '`..`' calls `food.eating` with `self` set to `health_food`, not the `food` class, so that `food.eating` affects `health_food`. This also allows changes to be made to any property, attribute, or property routine in a class, and that change will be reflected in all objects built from that class.

Example: Building a (More) Complex Object

At this point, enough material has been covered to develop a comprehensive example of a functional object that will serve as a summary of concepts introduced so far, as well as providing instances of a number of common properties from **`hugolib.h`**.

```
object woodcabinet "wooden cabinet"
{
     in emptyroom
     article "a"
     nouns "cabinet", "shelf", "shelves", \
          "furniture", "doors", "door"
     adjectives "wooden", "wood", "fine", "mahogany"

     short_desc
          "A wooden cabinet sits along one wall."
     when_open
          "An open wooden cabinet sits along
          one wall."
     long_desc
     {
          "The cabinet is made of fine mahogany wood,
          hand-crafted by a master cabinetmaker.  In
          front are two doors (presently ";
          if self is open
               print "open";
          else:  print "closed";
          print ")."
     }
     contains_desc
          "Behind the open doors of the cabinet you
          can see";           ! note semicolon--
                              ! no line feed

     key_object cabinetkey    ! a cabinetkey object
                              ! must also be created

     holding 0                ! starts off empty
     capacity 100

     before
     {
          object DoLookUnder
               {"Nothing there but dust."}
```

```
            object DoGet
                 {"The cabinet is far too heavy
                 to lift!"}
       }
       after
       {
            object DoLock
                 {"With a twist of the key, you lock the
                 cabinet up tight."}
       }

       is container, openable, not open
       is lockable, static
  }
```

Now, for bonus points for those who have looked ahead to *APPENDIX B: THE HUGO LIBRARY* to see what things like `when_open`, `contains_desc`, and `static` are for, how could the cabinet be converted into, say, a secret passage into another room?

The answer is: by adding a `door_to` property, such as:

```
door_to secondroom       ! a new room object
```

and

```
is enterable
```

as a new attribute. The cabinet can now be entered via: "go cabinet", "get into cabinet", "enter cabinet", etc.

Example: Building a Clock Event

Suppose that there is a clock object in a room. Here is a possible event:

```
event in clock
{
     local minutes, hours

     hours = counter / 60
     minutes = counter - (hours * 60)

     if minutes = 0
     {
          print "The clock chimes ";
```

```
            select hour
                 case 1:  print "one";
                 case 2:  print "two";
                 case 3:  print "three";
                 .
                 .
                 .
                 case 12:  print "twelve";[40]
            print " o'clock."
       }
  }
```

Whenever the player and the clock are in the same room (when a `runevents` command is given), the event will run.

Now, suppose the clock should be audible throughout the entire game—i.e., in any location on the game map. Simply changing the event definition to:

```
  event                    ! no object is given
  {
       ...
  }
```

will make the event a global one. (In this case, the `self` global is not altered.)

[40] You can actually use the `NumberWord` routine from **`hugolib.h`** to do this a lot more efficiently.

VI. FUSES, DAEMONS, AND SCRIPTS

VI.a. Introduction

While most of the previously discussed elements of programming with Hugo (such as events) are part of the internal architecture of the Hugo Engine, the means of running fuses, daemons, and scripts are written entirely in the Hugo language itself and contained in the Hugo Library.

VI.b. Fuses And Daemons

A *daemon* is the traditional name for a recurring activity. Hugo handles daemons as special events attached to objects that may be activated or deactivated (i.e., moved in and out of the scope of `runevents`). Since the daemon class is defined in the library, define a daemon itself using:

```
daemon <name>
{}
```

The body of the daemon definition is empty. The daemon object is only needed to attach the daemon event to, so the daemon definition must be followed by:

```
event [in] <name>
{
	...
}
```

Activate it by:

```
Activate(<name>)
```

which moves the specified daemon object into scope of the player. This way, whenever a `runevents` command is given (as it should be in the `Main` routine), the event attached to `<name>` will run.

Deactivate the daemon using:

```
Deactivate(<name>)
```

which removes the daemon object from scope. (It can be seen here that a daemon is actually a special type of object which is moved in and out of the scope of `runevents`, and that it is the event attached to the daemon that actually contains the code.)

A fuse is the traditional name for a timer—i.e., any event set to happen after a certain period of time. The fuse itself is a slightly more complex version of a daemon object, containing two additional properties as well as `in_scope`:

`timer`	the number of turns before the fuse event runs
`tick`	a routine that decrements `timer` and returns the number of turns remaining (i.e., the value of `timer`)

Similarly to a daemon, define a fuse in two steps:

```
fuse <name>
{}

event [in] <name>
{
	...
	if not self.tick
	{
		...
	}
}
```

and turn it on or off by:

```
Activate(<name>, <setting>)
```

and

```
Deactivate(<name>)
```

where `<setting>` is the initial value of the `timer` property.

Note that it is up to the event itself to run the `timer` and check for its expiration. The line

```
if not self.tick
```

runs the tick property—defined in the library, which is responsible for decrementing the timer—and executes the following conditional block if self.timer is 0.

VI.c. Scripts

Scripts are considerably more complex than fuses and daemons. The purpose of a script (also called a character script) is to allow an object—usually a character—to follow a sequence of actions turn-by-turn, independent of the player. Up to 16 scripts may be running at once.[41]

A script is represented by two arrays: scriptdata and setscript. The latter was named for programming clarity rather than for what it actually contains. Here's why:

To define a script, use the following notation:

```
setscript[Script(<char>, <num>)] = &CharRoutine, obj,
                                   &CharRoutine, obj,
                                   ...
```

(remembering that a hanging comma at the end of a line of code is a signal to the compiler that the line continues onto the next unbroken.)

Notice that "setscript" is actually an array, taking its starting element from the return value of the Script routine, which has <object> and <number> as its arguments.

Script returns a pointer within the large "setscript" array where <num> number of steps of a script for <object> may reside. A single script may have up to 32 steps. A step in a script consists of a routine and an object—both are required, even if the routine does not require an object. (Use the nothing object (0); see the CharWait routine in **hugolib.h** for reference.)

The custom in **hugolib.h** is that character script routines use the prefix "Char" although this is not required. Currently, routines provided include:

CharMove	(requiring a direction object)
CharWait	(using the nothing object)
CharGet	(requiring a takeable object)
CharDrop	(requiring an object held by the character)

[41] This is a library-set limit.

as well as the special routine

```
LoopScript        (using the nothing object)
```

which indicates that a script will continually execute. (It is the responsibility of the programmer to ensure that the ending position of the character or object is suitable to loop back to the beginning if `LoopScript` is used. That is, if the script consists of a complex series of directions, the character should always return to the same starting point.)

The sequence of routines and objects for each script is stored in the `setscript` array.

Scripts are run using the `RunScripts` routine, similar to `runevents`, the only difference being that `runevents` is an engine command while `RunScripts` is contained entirely in **hugolib.h**. The line:

```
RunScripts
```

will run all active object/character scripts, one turn at a time, freeing the space used by each once it has run its course.

Here is a sample script for a character named "Ned":

```
setscript[Script(ned, 4)] =   &CharMove, s_obj,
                              &CharGet, cannonball,
                              &CharMove, n_obj,
                              &CharWait, 0,
                              &CharDrop, cannonball
```

Ned will go south, retrieve the cannonball object, bring it north, wait a turn, and drop it. (The character script routines provided in the library are relatively basic; for example, `CharGet` assumes that the specified object will be there when the character comes to get it, so it's more or less up to the game author—at least when using the default library routines for character scripting—to have things well planned out.)

Other script-management routines in **hugolib.h** include:

`CancelScript(obj)`	to immediately halt execution of the script for <obj>
`PauseScript(obj)`	to temporarily pause execution of the script for <obj>
`ResumeScript(obj)`	to resume execution of a paused script

`SkipScript(obj)`	skips the script for `<obj>` during the next call to `RunScripts` only

The `RunScripts` routine also checks for `before` and `after` properties. It continues with the default action—i.e., the character action routine specified in the script—if it finds a false value.

To override a default character action routine, include a `before` property for the character object using the following form:

```
before
{
      actor CharRoutine
      {
            ...
      }
}
```

where `CharRoutine` is `CharWait`, `CharMove`, `CharGet`, `CharDrop`, etc.

VI.d. A Note About The **`event_flag`** Global

The library routines—particularly the `DoWait...` verb routines (invoked whenever a player types "`wait`", "`wait for (someone)`", or "`wait for 5 turns`"—expect the `event_flag` global variable to be set to a non-false value if something happens (i.e., in an event or script) so that the player may be notified and given the opportunity to quit waiting. For instance, the character script routines in **`hugolib.h`** set `event_flag` whenever a character does something in the same location as the player.

If **`hugolib.h`** is to be used, the convention of setting `event_flag` after every significant event should be adhered to.

VI.e. What Should I Be Able To Do Now?

Example: A Simple Daemon and a Simpler Fuse

The most basic daemon would be something like a sleep counter, which measures how far a player can go beginning from a certain rested state. Assume that the player's amount of rest is kept in a property called `rest`, which decreases by 2 each turn.

```
daemon gettired
{}

event in gettired
{
     player.rest = player.rest - 2
     if player.rest < 0
          player.rest = 0

     select player.rest
          case 20
               "You're getting quite tired."
          case 10
               "You're getting \Ivery\i tired."
          case 0
               "You fall asleep!"
}
```

Start and stop the daemon with `Activate(gettired)` and `Deactivate(gettired)`.

Now, as for a fuse, why not construct the most obvious example: that of a ticking bomb? (Assume that there exists another physical `bomb` object; `tickingbomb` is only the countdown fuse.)

```
fuse tickingbomb
{}

event in tickingbomb
{
     if not self.tick
     {
          if Contains(location, bomb)
```

```
                    "You vanish in a nifty KABOOM!"
               else
                    "You hear a distant KABOOM!"
               remove bomb
          }
     }
```

Start it (with a countdown of 25 turns):

```
     Activate(tickingbomb, 25)
```

and stop it with:

```
     Deactivate(tickingbomb)
```

VII. GRAMMAR AND PARSING

VII.a. Grammar Definition

Every valid player command must specified. More precisely, each usage of a particular verb must be detailed in full by the source code. Grammar definitions must *always* come at the start of a program, preceding any objects or executable code. That is, if several additional grammar files are to be included, or new grammar is to be explicitly defined in the source code, it must be done before any files containing executable code are included, or any routines, objects, etc. are defined.

The syntax used for grammar definition is:

```
[x]verb "<verb1>" [, "<verb2>", "<verb3>",...]
* <syntax specification 1>              <VerbRoutine1>
* <syntax specification 2>              <VerbRoutine2>
...
```

Now, what does that mean? Here are some examples from the library grammar file **`verblib.g`**:

```
verb "get"
     *                                        DoVague
     * "up"/"out"/"off"                       DoExit
     * "outof"/"offof"/"off" object           DoExit
     * "in"/"on" object                       DoEnter
     * multinotheld "from"/"off" parent       DoGet
     * multinotheld "offof"/"outof" parent    DoGet
     * multinotheld                           DoGet

verb "take"
     *                                        DoVague
     * "off" multiheld                        DoTakeOff
     * multiheld "off"                        DoTakeOff
     * multinotheld                           DoGet
```

```
        * multinotheld "from"/"off" parent      DoGet
        * multinotheld "offof"/"outof" parent   DoGet

    xverb "save"
        *                                       DoSave
        * "game"                                DoSave

    verb "read", "peruse"
        *                                       DoVague
        * readable                              DoRead

    verb "unlock"
        *                                       DoVague
        * lockable "with" held                  DoUnLock
        * lockable                              DoUnLock
```

Each `verb` or `xverb` header begins a new verb definition. An `xverb` is a special signifier that indicates that the engine should not call the `Main` routine after successful completion of the action. `xverb` is typically used with non-action, housekeeping-type verbs such as saving, restoring, quitting, and restarting.

Another thing that can be done is to specify:

```
    verb some_object
        * object                                DoVerb
```

which will have the effect of, instead of defining the verb with a dictionary word, checking at runtime `some_object.noun` as the verb word to be matched. What this allows is for the `some_object.noun` property to be a routine that can return varying values at runtime in order to provide for dynamic grammar, if required. However, since this sort of dynamic grammar isn't often required, static grammar definitions are far more common.

Next in the header comes one or more verb words. Each of the specified words will share the following verb grammar *exactly*. This is why "`get`" and "`take`" in the above examples are defined separately, instead of as

```
    verb "get", "take"
```

In this way, the commands like

```
>get up
```

and

```
>take off hat
```

are allowable, while

```
>take up
```

and

```
>get off hat
```

won't make any sense.

Each line beginning with an asterisk ('*') is a separate valid usage of the verb being defined. (Every player input line must begin with a verb. Exceptions, where a command is directed to an object as in

```
>Ned, get the ball
```

will be dealt with later.)

Up to two objects and any number of dictionary words may make up a syntax line. The objects must be separated by at least one dictionary word.

Valid object specifications are:

`object`	any visible object (the direct object)
`xobject`	the indirect object
`attribute`	any visible object that is `attribute`
`parent`	an xobject that is the parent of the object
`held`	any object possessed by the player object
`notheld`	an object explicitly not held
`anything`	any object, held or not, visible or not
`multi`	multiple visible objects
`multiheld`	multiple held objects
`multinotheld`	multiple notheld objects
`number`	a positive integer number
`word`	any dictionary word
`string`	a quoted string
`(RoutineName)`	a routine name, in parentheses
`(objectname)`	a single object name, in parentheses

(If a number is specified in the grammar syntax, it will be passed to the verbroutine in the `object` global. If a string is specified, it will be passed in the

engine's `parse$` variable, which can then be turned into a string array using the `string` function.)

Dictionary words that may be used interchangeably are separated by a slash ('/').

Two or more dictionary words in sequence must be specified separately. That is, in the input line:

```
>take hat out of suitcase
```

the syntax line

```
* object "out" "of" container
```

will be matched, while

```
* object "out of" container
```

would never be recognized, since the engine will automatically parse "out" and "of" as two separate words; the parser will never find a match for "out of".

Regarding object specification within the syntax line: once the direct object has been found, the remaining object in the input line will be stored as the xobject. That is, in the example immediately above, a valid object in the input line with the attribute `container` will be treated as the indirect object by the verb routine.

Note: An important point to remember when mixing dictionary words and objects within a syntax line is that, unless directed differently, the parser may confuse a word-object combination with an invalid object name.

Consider the following:

```
verb "pick"
     * object                 DoGet
     * "up" object            DoGet
```

This definition will result in something like

```
>pick up box
You haven't seen any "up box", nor are you likely to
in the near future even if such a thing exists.
```

(assuming that "up" has been defined elsewhere as part of a different object name, as in **objlib.h**), because the processor processes the syntax

```
* object
```

and determines that an invalid object name is being used; it never gets to

```
* "up" object
```

The proper verb definition would be ordered like

```
verb "pick"
     * "up" object              DoGet
     * object                   DoGet
```

so that both "`pick <object>`" and "`pick up <object>`" are valid player commands. It's generally good practice to make sure that more specific grammar precedes more general grammar for this reason.

To define a new grammar condition that will take precedence over an existing one—such as in **verblib.g**—simply define the new condition first (i.e., before including **verblib.g**).

Note: As a rule, unless you need to preempt the library's normal grammar processing, include any new grammar *after* the library files. (The reason for this is that the library grammar is carefully tuned to handle situations exactly like that described above.)

A single object may be specified as the only valid object for a particular syntax:

```
verb "rub"
     * (magic_lamp)             DoRubMagicLamp
```

will produce a "`You can't do that with...`" error for any object other than the `magic_lamp` object.

Using a routine name to specify an object is slightly more involved: the engine calls the given routine with the object specified in the input line as its argument; if the routine returns true, the object is valid—if not, a parsing error is expected to have been printed by the routine. If two routine names are used in a particular syntax, such as

```
* (FirstRoutine) "with" (SecondRoutine)
```

then `FirstRoutine` validates the object and `SecondRoutine` validates the xobject.

VII.b. The Parser

Immediately after an input line is received, the engine calls the parser, and the first step taken is to identify any invalid words, i.e., words that are not in the dictionary table.

Note: One non-dictionary word or phrase is allowed in an input line, providing it is enclosed in quotation marks. If the command is successfully parsed and the quoted word or phrase is matched to a `string` grammar token, that string is passed to `parse$`. More than one non-dictionary word or phrase (even if the additional phrases are enclosed in quotes) are not allowed.

The next step is to break the line down into individual words. Words are separated by spaces and basic punctuation (including "!" and "?") which are removed. All characters in an input line are converted to lowercase (except those inside quotation marks).

The next step is to process the four types of special "words" which may be defined in the source code.

Removals are the simplest. These are simply words that are to be automatically removed from any input line, and are generally limited to words such as "a" and "the" which would, generally speaking, only make grammar matching more complicated and difficult. The syntax for defining a removal is:

```
removal "<word1>"[, "<word2>", "word<3>",...]
```

as in

```
removal "a", "an", "the"
```

Punctuation is similar to a removal, except it specifies the removal of individual characters instead of whole words:

```
punctuation "<character1>[<character2>...]
```

as in

```
punctuation "$%"
```

Synonyms are slightly more complex. These are words that will never be found in the parsed input line; they are replaced by the specified word for which they are a synonym.

```
synonym "<synonym>" for "<word>"
```

as in

```
synonym "myself" for "me"
```

The above example will replace every occurrence of "myself" in the input line with "me". Usage of synonyms will likely not be extensive, since of course it is possible to, particularly in the case of object nouns and adjectives specify synonymous words which are still treated as distinct.

Compounds are the final type of special word, specified as:

```
compound "<word1>", "<word2>"
```

as in

```
compound "out", "of"
```

so that the input line

```
>get hat out of suitcase
```

would be parsed to

```
>get hat outof suitcase
```

Depending on the design of grammar tables for certain syntaxes, the use of compounds may make grammar definition more straightforward, so that by using the above compound,

```
verb "get"
     * multinotheld "outof"/"offof"/"from" parent
```

is possible, and likely more desirable to

```
verb "get"
     * multinotheld "out"/"off" "of" parent
     * multinotheld "from" parent
```

When the parser has finished processing the input line, the result is a specially defined (by the Hugo Engine) array called `word[]`, where the number of valid elements is held in the global variable `words`. Therefore, in

```
>get the hat from the table
```

the parser—using the removals defined in **`hugolib.h`**—will produce the following results:

```
word[1] = "get"
word[2] = "hat"
word[3] = "from"
word[4] = "table"

words = 4
```

Note: Multiple-command input lines are also allowed, provided that the individual commands are separated by a period (".").

```
>get hat.  go n.  go e.
```

would become

```
word[1] = "get"
word[2] = "hat"
word[3] = ""
word[4] = "go"
word[5] = "n"
word[6] = ""
word[7] = "go"
word[8] = "e"
word[9] = ""

words = 9
```

(See **`hugolib.h`** for an example of how

```
>get hat then go n
```

is translated into:

```
word[1] = "get"
word[2] = "hat"
```

```
word[3] = ""
word[4] = "go"
word[5] = "n"
```

in the `Parse` routine.)

A maximum of thirty-two words is allowed. The period is in each case converted to the empty dictionary entry (“”; dictionary address = 0), which is a signal to the engine that processing of the current command should end here.

Note: The parsing and grammar routines also recognize several system words, each in the format “`~word`”. These are:

`~and`	**referring to:**	**multiple specific objects**
`~all`	**“ “**	**multiple objects in general**
`~any`	**“ “**	**any one of a list of objects**
`~except`	**“ “**	**an excluded object**
`~oops`	**to correct an error in the previous input line**	

To allow an input line to access any of these system words, a synonym must be defined, such as:

```
synonym "and" for "~and"
```

The library defines several such synonyms.

VII.c. What Should I Be Able To Do Now?

It should by now be relatively straightforward how to go about adding a new verb (with appropriate grammer)—or even modifying an existing one. For instance, consider a game in which disco dancing plays an absolutely vital role, and where the command "`>GET DOWN`" must at all costs be implemented as a synonym for "`>DANCE`" or "`>BOOGIE`".

For starters, you'll need to add the initial grammar and verbroutine:

```
verb "dance", "boogie"
     *                        DoDance
```

and

```
routine DoDance
{
     "You get down, all night long."
}
```

Keep in mind that the verb definition, as with all grammar, must come before any other code, definitions, etc. Now, you'll have to add the "`>GET DOWN`" grammar:

```
verb "get"
     * "down"                 DoDance
```

Now, this must come both before any other code or definitions as well as the *existing* grammar for "`>GET <object>`" (from `VERBLIB.G`). Otherwise, the regular grammar for

```
     * object                 DoGet
```

will take precedence. By superseding it, however, we ensure that any grammar matching the desired pattern will result in `DoDance` being called instead.

VIII. JUNCTION ROUTINES

VIII.a. Before We Get To The Routines

Because, the engine is unaware of such things as attributes, properties, and objects in anything but a technical sense[42], there are provided a number of routines to facilitate communication between the engine and the program proper. Along with these junction routines are certain global variables and properties that are pre-defined by the compiler and accessed directly by the engine. They are:

GLOBALS

`object`	the direct object of a verb
`xobject`	the indirect object
`self`	self-referential object
`words`	total number of words
`player`	the player object
`actor`	the player, or character obj. (for scripts)
`location`	location of the player
`verbroutine`	the verb routine address
`endflag`	if not false (0), call `EndGame`
`prompt`	for the player input line
`objects`	total number of objects
`system_status`	after certain operations

PROPERTIES

`name`	basic object name
`before`	pre-verb routines
`after`	post-verb routines
`noun`	noun(s) for referring to object

[42] In other words, it is the library that defines all the rules and useful-sounding names for properties, routines, and the like; the engine doesn't really have any idea about the higher-level work being done by the library.

adjective	adjective(s) for referring to object
article	"a", "an", "the", "some", etc.

(Additionally, the aliases nouns and adjectives for noun and adjective, respectively, are defined by the library.)

Junction routines are not required. The engine has built-in default routines, although it's likely that not all of these will be satisfactory for most programmers. **hugolib.h** contains each of the following routines which fully implement all the features of the library. If a different routine is desired in place of a provided one, the routine should be substituted using replace.

VIII.b. Parse

The Parse routine, if one exists, is called by the engine parser. Here, the program itself may modify the input line before grammar matching is attempted. What happens is:

1. The input line is split into discrete words (by the engine).
2. The Parse routine, if it exists, is called.
3. Control returns to the engine for grammar matching.
4. During grammar matching, the FindObject routine may be called (possibly repeatedly).

For example, the Parse routine in **hugolib.h** takes care of such things as pronouns ("he", "she", "it", "them") and repeating the last legal command (with "again" or simply "g").

Returning true from the Parse routine calls the engine parser again (i.e., returns to step 1 in the process above); returning false continues normally. This is useful in case the Parse routine has changed the input line substantially, requiring a reconfiguration of the already split words.

The HugoFix debugging library can be used at runtime to monitor the goings-on of the Parse routine by enabling parser monitoring with the "$pm" command.[43]

[43] For more information on debugging using HugoFix, see *APPENDIX D: HUGOFIX AND THE HUGO DEBUGGER*.

Note: Since the library's `Parse` routine is rather extensive, a provision is made for a `PreParse` routine—which in the library is defined as being empty—which may more easily be replaced for additional parsing.

VIII.c. ParseError

The `ParseError` routine is called whenever a command is invalid. `ParseError` is called in the form:

```
ParseError(<errornumber>, <object>)
```

where `<object>` is the object number (if any) of the object involved in the error.

Note: The engine also sets up the special variable `parse$`, which represents the illegal component of an input line, whether it is the verb itself, an object name, a partial object name, or any other word combination.

For example:

```
print "The illegal word was:  "; parse$; "."
```

The default responses provided by the engine parse error routine are:

ERROR NUMBER	RESPONSE
0	"What?"
1	"You can't use the word <parse$>."
2	"Better start with a verb."
3	"You can't <parse$> multiple objects."
4	"Can't do that."
5	"You haven't seen any <parse$>, nor are you likely to in the near future even if such a thing exists."

6	`"That doesn't make any sense."`
7	`"You can't use multiple objects like that."`
8	`"Which <parse$> do you mean,...?"`
9	`"Nothing to <parse$>."`
10	`"You haven't seen anything like that."`
11	`"You don't see that."`
12	`"You can't do that with the <parse$>."`
13	`"You'll have to be a little more specific."`
14	`"You don't see that there."`
15	`"You don't have that."`
16	`"You'll have to make a mistake first."`
17	`"You can only correct one word at a time."`

The `ParseError` routine in **hugolib.h** provides customized responses that take into account such things as, for example, whether the player is first or second-person, whether or not an object is a character or not, and if so, if it is male or female, etc.

If the `ParseError` routine does not provide a response for a particular `<errornumber>`, it should return false. Returning false is a signal that the engine should continue with the default message. Returning 2 is a signal to reparse the entire existing line (useful in cases where a peculiar syntax is trapped as an error, changed, and must then be reparsed).

Note: If custom error messages are desired for user parsing routines, replace the routine `CustomError` with a new routine (called with the same parameters as `ParseError`), providing that `<errornumber>` is greater than or equal to 100.

VIII.d. EndGame

The `EndGame` routine is called immediately whenever the global variable `endflag` is non-zero, regardless of whether or not the current function has not yet been terminated.

`hugolib.h`'s `EndGame` routine behaves according to the value to which `endflag` is set:

endflag	**RESULT**
1	Player wins
2	Player's demise
0	Other ending—not provided for by default `PrintEndGame` routine)

Returning false from `Endgame` terminates the game completely; returning non-false restarts.

Note: To modify only the message displayed at the end of the game (defaults: "`*** YOU'VE WON THE GAME! ***`" and "`*** YOU ARE DEAD ***`"), replace the `PrintEndGame` routine. Other than being non-false, the various values of `endflag` are insignificant except to `PrintEndGame`.

VIII.e. FindObject

The `FindObject` routine takes into account all the relevant properties, attributes, and object hierarchy to determine whether or not a particular object is available in the current context. For example, the child of a parent object may be available if the parent is a platform, but unavailable if the parent is a container (and closed)—although internally, the object hierarchy is the same. `FindObject` is called via:

```
FindObject(<object>, <location>)
```

where <object> is the object in question, and <location> is the object where its availability is being tested. (Usually <location> is a room, unless a different parent has been specified in the input line.)

FindObject returns true (1) if the object is available, false (0) if unavailable. It returns 2 if the object is visible but not physically accessible.

The FindObject routine in **hugolib.h** considers not only the location of <object> in the object tree, but also tests the attributes of the parent to see if it is open or closed. As well, it checks the found_in property, in case <object> has been assigned multiple locations instead of an explicit parent, and then scans the in_scope property of the object (if one exists).

Finally, the default behavior of the library's FindObject requires that a player have encountered an object for it to be valid in an action, i.e., it must have the known attribute set. To override this, replace the routine ObjectisKnown with a routine that returns an unconditional true value.

There is one special case in which the engine expects the FindObject routine to be especially helpful: that is if the routine is called with <location> equal to 0. This occurs whenever the engine needs to determine if an object is available *at all*—regardless of any rules normally governing object availability—such as when an anything grammar token is encountered, or the engine needs to disambiguate two or more seemingly identical objects. (Also, FindObject may be called by the engine with both <object> and <location> equal to 0 to reset any library-based object disambiguation.)

The HugoFix debugging library can be used at runtime to monitor calls to FindObject by enabling the "$fi" command.[44]

VIII.f. SpeakTo

The SpeakTo routine is called whenever an input line begins with a valid object name instead of a verb. This is so the player may direct commands to (usually) characters in the game. For example:

```
>Professor Plum, drop the lead pipe
```

It is up to the SpeakTo routine to properly interpret the instruction. SpeakTo is called via:

```
SpeakTo(<character>)
```

[44] For more information on debugging using HugoFix, see *APPENDIX D: HUGOFIX AND THE HUGO DEBUGGER*.

where <character> in the above example would be the Professor Plum object. The globals object, xobject, and verbroutine are all set up as normal. For the above example, then, these would be

```
object          leadpipe
xobject         nothing
verbroutine     &DoDrop
```

when SpeakTo is called.

hugolib.h's SpeakTo routine provides basic interpretation of questions, so that

```
>Professor Plum, what about the lead pipe?
```

may be directed to the proper verb routine, as if the player had typed:

```
>ask Professor Plum about the lead pipe
```

Imperative commands are, such as

```
>Colonel Mustard, stand up
```

are first directed to the order_response property of the character object in question. It is subsequently up to <character>.order_response to analyze verbroutine (as well as object and xobject, if applicable) to see if the request is a valid one. If no response is provided, order_response should return false.

The HugoFix debugging library can be used at runtime to monitor calls to SpeakTo by enabling the "$pm" command.[45]

```
order_response
{
     if verbroutine = &DoGet
          "I would, but my back is too sore."
     else
          return false
}
```

Note: It is important to check in an order_response property if any objects to be acted upon are present (or otherwise available), since this check is not necessarily done before SpeakTo is called.

[45] For more information on debugging using HugoFix, see *APPENDIX D: HUGOFIX AND THE HUGO DEBUGGER*.

When something like the following is directed toward a character:

```
>BOB, GET THE PACKAGE
```

`SpeakTo(bob)` will be called with `verbroutine = &DoGet` and `object = package`, even if the `package` object is not physically present.

VIII.g. Perform

The `Perform` routine is what is called by the engine in order to execute the appropriate verbroutine with the given object(s) and/or indirect object, if either or both are applicable. It is the responsibility of `Perform` to do the appropriate checking of `before` routines to determine if execution actually gets to the verbroutine. `Perform` is called as:

```
Perform(<verbroutine>, <object>, <xobject>, <queue>,
     <isxverb>)
```

The first three arguments represent the match verb (always), object (if given), and indirect object, i.e., the xobject (if given). The `<queue>` is 0 unless the verbroutine is being called more than once for multiple objects. (As a special case, `<queue>` is –1 if `object` or `xobject` is a number supplied in the input as one or more digits, in order to signal `Perform` not to do normal `before`/`after` routine calling.) The `<isxverb>` argument is true if the grammar for invoking `Perform` designates an xverb[46].

For example, various player commands might (approximately, depending on verbroutine and object names) result in the routine calls:

```
>i
Perform(&DoInventory, 0, 0, 0)

>get key
Perform(&DoGet, key_object, 0, 0)

>put the key on the table
Perform(&DoGet, key_object, 0, 0)

>turn the dial to 127
Perform(&DoTurn, dial, 127, -1)
```

[46] The `<isxverb>` argument is new in v3.1.

```
>get key and banana
Perform(&DoGet, key_object, 0, 1)
Perform(&DoGet, banana, 0, 2)
```

(If no `Perform` routine exists, the engine performs a default calling of `player.before`, `location.before`, `xobject.before`, `object.before`, and finally `verbroutine` if none of those returns true.)

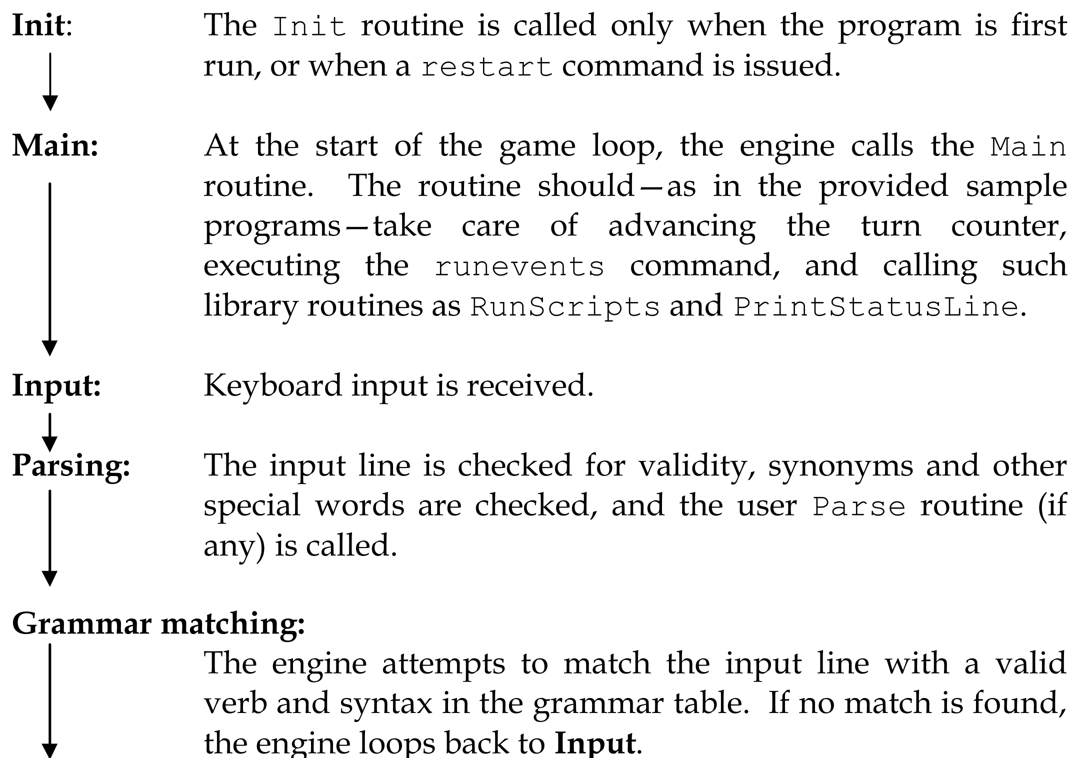
Using HugoFix's parser monitoring ("`$pm`") at runtime will trace calls to `Perform`.[47]

[47] For more information on debugging using HugoFix, see *APPENDIX D: HUGOFIX AND THE HUGO DEBUGGER*.

IX. THE GAME LOOP

IX.a. Overview Of The Game Loop

This the basic execution pattern that the Hugo Engine follows during program execution. (Also mentioned are the calling of `before` routines and the `verbroutine` by `Perform` in **`hugolib.h`**. While not necessarily part of the game loop—since they may or may not be included in a program—they are mentioned here because they are relevant to any Hugo program that uses the standard Hugo Library.)



Init: The `Init` routine is called only when the program is first run, or when a `restart` command is issued.

Main: At the start of the game loop, the engine calls the `Main` routine. The routine should—as in the provided sample programs—take care of advancing the turn counter, executing the `runevents` command, and calling such library routines as `RunScripts` and `PrintStatusLine`.

Input: Keyboard input is received.

Parsing: The input line is checked for validity, synonyms and other special words are checked, and the user `Parse` routine (if any) is called.

Grammar matching: The engine attempts to match the input line with a valid verb and syntax in the grammar table. If no match is found, the engine loops back to **Input**.

Otherwise, a successful grammar match results in at least the `verbroutine` global being set, as well as potentially `object` and `xobject`.

Before routines (as called by `Perform` in `hugolib.h`):

If any objects were specified in the input line, their `before` properties are checked in the following order, for each object:

```
player.before
location.before
xobject.before        (if applicable)
object.before         (if applicable)
```

If any of these property routines returns true, the engine skips the verb routine. (The `react_before` property for relevant objects is checked at this time as well.)

Verb routine (as called by `Perform` in `hugolib.h`):

If no before property routine returns true, the verb routine is run.

If an action is successfully completed, the verb routine should return true. Returning false negates any remaining commands in the input line.

`Perform` does not run any `after` property routines for `object` or `xobject`; that is up to the verb routine. It does run both `player.after` and `location.after` if the verbroutine returns true. (The `react_after` property for relevant objects is checked at this time as well.)

(Control returns from the library `Perform` routine to the engine)

When finished, the engine loops back to **Main**, calling the `Main` routine only if the last verb matched was not an xverb.

Note: Setting the global endflag at any point to a non-zero value will terminate the game loop and run the `EndGame` junction routine.

IX.b. What Should I Be Able To Do Now?

By this point, you've been introduced to the basic facilities through which the Hugo Engine communicates with a running Hugo game: the junction routines `EndGame`, `FindObject`, `Parse`, `ParseError`, `Perform`, and `SpeakTo`. Becoming familiar with their implementation and use (and even inner workings) is an important step toward mastering an understanding of the Hugo game loop, including determining how a player input line is parsed, what objects are available or in scope, and how a command is either dispatched to a verbroutine or directed to another character. You should be able to create your own verbroutines (and grammar) to handle actions not provided by default in the library. You should now understand how to create an `order_response` property for characters to respond to actions passed to them by `SpeakTo`.

It should be apparent how a game can implement custom versions of things like end-of-game behavior, parser messages, etc. without editing the library itself by using the compiler's `replace` directive with library routines such as `EndGame` and `ParseError`. Most often, a programmer will copy the selected routine out of **`hugolib.h`** (or wherever it comes from) and paste it into the game's source, changing (for instance):

```
routine EndGame ...
```

to

```
replace EndGame ...
```

and customize the `EndGame` routine's messages, behavior, or whatever the desired modifications may involve.

For instance, to change the default parser error message

```
You don't need to use the word "<unknown word>".
```

to something along the lines of

```
[This game does not recognize "<unknown word>".]
```

first copy the `ParseError` routine from **`hugolib.h`** to the game's source, and change, in the copied `ParseError`,

```
routine ParseError(errornumber, object)
```

to

```
replace ParseError(errornumber, object)
```

and modify the case for the error message in question from

```
case 1
     print CThe(player); \
          MatchPlural(player, "doesn't, "don't"); \
          need to use the word \""; \
          parse$; "\"."
```

to

```
case 1
     print "[This game does not recognize \""; \
          parse$; "\".]"
```

You should also by now have an understanding of how to override the game loop using `before` and `after` routines in order to provide for custom responses and/or behavior not directly provided by the library. Normally, for example, an object having the `static` attribute is automatically treated as untakeable by the library. But what if you created a heavy boulder and wanted to have the response

```
The boulder is far too heavy to lift!
```

in place of the library's default "`You can't take that.`" message? You would simply interrupt the `DoGet` routine before it even executes via a `before` property on the boulder object:

```
before
{
     object DoGet
     {
          "The boulder is far too heavy to lift!"
     }
}
```

X. USING THE OBJECT LIBRARY

The Hugo Object Library (**objlib.h**), included by default by **hugolib.h** as part of the standard Hugo Library, provides a number of useful classes for common elements of many games. These classes can be used as-is to create objects or as base classes for more complex and/or game-specific classes.

X.a. Rooms and Directions

Most games will make use of rooms, directions, and possibly characters. A room in this context is an object which specifically functions as a location in the game world, and as such contains other objects as children in branches beneath it in the object tree. Despite being called "rooms", these generic locations aren't explicitly indoors or outdoors; most games won't make a distinction except in the textual description (the `long_desc`) of the location.

A room is defined like this:

```
room living_room "living room"
{
     long_desc
          "The living room is about fifteen feet
          square, with a bay window on one wall
          looking out over the garden.  The kitchen
          is to the south, or you can walk into the
          back hallway to the east."
}
```

Since a room is not placed within another object in the object tree (using the `in` directive), it automatically has a parent of 0 or `nothing`. By inheriting from the `room` class, the `living_room` object acquires the characteristics defined in the object library, including being `static` and `light`[48].

[48] This also means that in order to create a dark location—i.e., one that has no intrinsic lighting—it is necessary to put an explicit "`not light`" in the object's definition. In that case, in order for the player to see anything in the location, light will have to be provided either by an object in the location (such as a lamp) or by something portable that can be brought into the location (like a flashlight).

Additionally, as with other classes in the object library, the `type` property of an object can be checked to see which class it was derived from. In this case, `living_room.type` would equal `room`. (Of course, `living_room` or an object subsequently derived from `living_room` could set its `type` property to something other than `room`.)

Travel between rooms is managed by default using the eight cardinal compass directions (north, northeast, east, southeast, south, southwest, west, northwest) as well as up, down, in, and out. These are represented in object form as `n_obj`, `ne_obj`, `e_obj`, `se_obj`, `s_obj`, `sw_obj`, `w_obj`, `nw_obj`, `u_obj`, `d_obj`, `in_obj`, and `out_obj` (each derived from the `direction` class). Each of these objects is defined in the object library as a child of the `compass` parent object, and each defines an appropriate `dir_to` property reflecting the direction of travel it describes. For instance, `n_obj.dir_to` equals `n_to`, `in_obj.dir_to` equals `in_to`, etc. It is the `dir_to` property of a direction that is used to map out travel from one location to another.

The `living_room` object's description claims that the kitchen is to the south and the back hallway is to the east, but neither connection is known to the game until those directional links are added to the object definition (and, of course, assuming that the `kitchen` and `back_hallway` objects also exist).

```
room living_room "living room"
{
     long_desc
          "The living room is about fifteen feet
          square, with a bay window on one wall
          looking out over the garden.  The kitchen
          is to the south, or you can walk into the
          back hallway to the east."
     s_to kitchen
     e_to back_hallway
}
```

The player will now be able to go either south or east from the kitchen. Keep in mind, however, that it is possible to make travel from one location to another one-way. In order to allow the player to travel back to the living kitchen in the same direction, you would need to add

```
n_to living_room
```

to the `kitchen` object and

```
w_to living_room
```

to the `back_hallway` object.

X.b. Characters

The basic `character` class takes care of defining the basic elements that the library expects character objects to have. These include obvious attributes such as `living`, as well as useful properties such as `capacity` and `holding` for carrying objects. Also importantly, the `character` class defines its `pronouns` property as:

```
pronouns "he", "him", "his", "himself"
```

The accompanying `female_character` class (which is identical to the `character` class but with the `female` attribute) defines:

```
pronouns "she", "her", "her", "herself"
```

The order of pronouns is *subject*, *object*, *possessive*, *reflexive*, so that the library can refer to the `pronouns` property to print appropriate messages such as:

```
print capital object.pronoun; " is in the room."
```

```
He is in the room.
```

```
print "Bob gave "; object.pronoun #2; " the box. "
```

```
Bob gave him the box.
```

```
print capital object.pronoun; " opened ";
object.pronoun #3; " box. "
```

```
He opened his box.
```

```
print capital object.pronoun; " looked at ";
object.pronoun #4; " in the mirror. "
```

```
He looked at himself in the mirror.
```

A `player_character` class is also provided that will usually be the basis for the player character (PC). A game's PC can be defined easily as:

```
player_character you "you"
{}
```

The PC is by default named in the second person (as opposed to the first-person "I" or the third-person "he", "she", "it", or "them"). To change to another form, it will be necessary to redefine the PC's `pronouns` property accordingly, as well as to change the global variable `player_person`[49] (which defaults to 2) to 1 or 3, as appropriate, and optionally to give a plural PC the `plural` attribute (in the rare case where that may be desired).

X.c. Character responses

One thing that will likely be important for NPCs (non-player characters) is enabling them to respond to questions and otherwise interact with the player. This is traditionally accomplished by implementing NPC responses to the verbroutines `DoAsk`, `DoTell`, `DoShow`, and `DoGive`.

```
>ASK GUSTAV ABOUT APPLE
"I must admit I rather prefer them to bananas," Gustav
tells you.

>GIVE BANANA TO GUSTAV
"No, thank you," Gustav says.  "I would rather have an
apple."
```

The grammar for asking an NPC about something looks something like this:

```
verb "ask"
     * living about xobject         DoAsk
```

The object is the NPC being asked, the xobject is whatever is being asked about, and the verbroutine is `DoAsk`. The response is handled in the NPC's `after` property routine:

```
after
{
     object DoAsk
     {
          select xobject
          case apple
               "\"I must admit I rather prefer them to
               bananas,\" Gustav tells you."
```

[49] The library uses the `player_person` global to properly format messages to the player.

```
            case else
                return false   ! important
        }
    }
```

Note that it's necessary to return false if the routine fails to find an appropriate response.

`DoTell` responses are handled similarly to `DoAsk`, since the NPC is the object and whatever is being told about is the xobject:

```
    verb "tell"
        * living about xobject        DoTell
```

DoGive and DoShow, however, are handled differently, since the word ordering is different:

```
    verb "give"
        * object to living            DoGive

    verb "show"
        * object to living            DoShow
```

For these, the `after` property routine will look something like:

```
    after
    {
        object DoAsk
        {
            ...
        }
        xobject DoGive
        {
            select object
            case banana
                "\"No thank you,\" Gustav says.  \"I
                would rather have an apple.\""
            case else
                return false
        }
        xobject DoShow
        {
            select object
            ...
```

```
          case else
               return false
     }
}
```

X.d. Scenery and Components

It has become more and more expected of interactive fiction that objects mentioned in the textual description of a location should be implemented in a manipulable fashion. With this goal in mind, something like the following would be less than desirable:

```
PRISON CELL
  The entire place is probably just shy of fifty
square feet.  The bars on the doors and single small
window ensure that you won't be going anywhere anytime
soon.

>EXAMINE WINDOW
You don't need to use the word "window".
```

Depending on the game (and, arguably, the player) that response may be somewhat jarring in light of the window just mentioned in the room description. It may be gotten around by adding an embellishment like this:

```
scenery prison_window "prison window"
{
     in prison_cell
     article "a"
     adjectives "single", "small", "prison", "cell"
     noun "window"
}
```

The most important characteristic of a scenery object created using the `scenery` class is that it will not be listed by the library as part of the room's contents (in this case, the contents of `prison_cell`). The `scenery` class is otherwise relatively unadorned: looking at a scenery object will produce a generic message about seeing nothing special, attempting to take a scenery object will generate a generic "You can't take that"-type response, etc.[50] The scenery object can be fleshed out with a `long_desc` property and `before`/`after` handling for desired verbroutines.

[50] The `scenery` class has the `static` attribute, which makes scenery objects untakeable. This is largely the point of scenery objects.

Components are similar to scenery objects in two important respects in that they're not intended to be taken and they're not specifically listed in any itemization of contents. Consider a case where a game might contain a machine (for fun, a nefarious machine) and a lever. The intention might be that the lever can be manipulated separately from the nefarious machine itself so that ">PULL LEVER" elicits a different response than simply ">PULL MACHINE". At the same time, however, something like the following is probably undesirable:

```
A nefarious machine whirs and buzzes in the corner.
There is also a lever here.
```

What is needed is a way to implement the lever as a separate though inseparable part of the nefarious machine object. The `component` class provides for this.

```
component lever "lever"
{
     part_of nefarious_machine
     article "a"
     noun "lever"
}
```

The `part_of` property specifies the primary object (the nefarious machine) of which this object (the lever) is a component; it is not necessary to place the component object *in* the primary object; in fact, doing so will probably lead to all manner of extra complications especially if the primary object isn't a container or platform, isn't open, etc. A component object will automatically be available to the player whenever the primary object is.

X.e. Doors

Doors are fairly common objects, and a given game—particularly one with a significant number of indoor locations—will likely make frequent use of them. The unfortunate thing is that they can be somewhat finicky and repetitive to code, ensuring that they respond to opening, closing, locking (if applicable), providing an appropriate open or closed description, and behaving appropriately from either side. The object library's `door` class provides a simple implementation that will largely suffice for most basic doors.

Here's how to put a simple door between the `kitchen` and `living_room` locations created above:

```
door kitchen_door "kitchen door"
{
```

```
    between kitchen, living_room
    article "the"
    adjective "kitchen"
    noun "door"
}
```

The `between` property takes care of making the room available in both locations as well as determining where the player travels to when leaving either location. In order to incorporate the door into the `kitchen` and `living_room` locations, it's only necessary to change the two room objects to specify:

```
n_to
{
     return kitchen_door.door_to
}
```

for the `kitchen` object and

```
s_to
{
     return kitchen_door.door_to
}
```

for the `living_room` object. Notice that the use of `kitchen_door.door_to` is the same for both; the `door` class's `door_to` property returns the appropriate location from the `between` property depending on where the player is when the `door_to` property is checked. The `door_to` property will also automatically result in an attempt to open a closed door (by calling the `DoOpen` verbroutine), resulting in an additional turn by calling the `Main` routine.

X.f. Vehicles

Less frequently used but somewhat more complex than doors are vehicles. Anything from a car to a UFO to a wild zebra may make an appearance in a game, and often it is necessary that the player be able to use that object—whatever it may be—as a means of moving around the game's geography. The object library's `vehicle` class provides a generic class that can be used to implement any of these (just for starters), allowing behavior like the following:

```
>GET ON THE HORSE
You get on the wild mustang.

>RIDE WEST
```

```
Dusty Trail
  This trail leads southwest out of town toward the
river valley and the old prospector's camp.
```

Note: Before using vehicle objects it is necessary to set the compiler flag `USE_VEHICLES`.

Create a vehicle from the `vehicle` class like this:

```
vehicle mustang "wild mustang"
{
     article "a"
     adjectives "wild", "untamed"
     nouns "mustang", "horse"
     vehicle_verb "ride"
     preposition "on", "off"
}
```

The `vehicle_verb` property provides one or more synonyms for the verb used to "drive" this particular vehicle object. In the case of a horse, it is appropriately "ride". The `preposition` property defaults in the `vehicle` class itself defaults to "in" and "out", but in the case of a horse should be changed to the more suitable "on" and "off".

It is also necessary to provide grammar to relate the words in the `vehicle_verb` list to the object library's `DoMoveInVehicle` routine.Grammar he following is recommended:

```
verb "<verb1>"[, "<verb2>",...]
*                        DoMoveinVehicle
* object                 DoMoveinVehicle
```

So, for our horse "vehicle", something like the following might suffice:

```
verb "ride
*                        DoMoveinVehicle
* object                 DoMoveinVehicle
```

It is possible to easily maintain some control over whether a vehicle is currently capable of moving via the `vehicle_move` property. This property, which is true by default, can return false (after printing an appropriate failure message) if the vehicle is currently not capable of being driven (or ridden or sailed or whatever the appropriate action may be).

To prevent the player from riding the mustang until the horse has been fed, implement a `vehicle_move` property similar to this:

```
vehicle_move
{
     if self is not fed  ! assuming a 'fed' attribute
     {
          "This horse isn't going anywhere until you
          feed it."
          return false
     }
     else
          return true
}
```

And finally, it is also necessary to give the vehicle some idea about where it is able to move. Every location that a vehicle may travel to must contain the vehicle in a `vehicle_path` property. For instance, a location to which both the mustang and a `wagon` object may move would need:

```
vehicle_path mustang, wagon
```

X.g. Plural and Identical Objects

Sometimes it is desirable to have a player be able to (or required to) refer to multiple objects as a group, or to be able to refer to only a certain number of such objects out of a larger group even if all the objects are identical. The object library's `plural_class` and `identical_class` make these sorts of things possible.

Note: Before using plural or identical objects it is necessary to set the compiler flag `USE_PLURAL_OBJECTS` and call `InitPluralObjects` (usually in the `Init` routine).

The `plural_class` is used in situations where two or more similar objects may be grouped together and referred to as a unit. For instance:

```
There are a fudge sundae and a butterscotch sundae
here.

>GET BUTTERSCOTCH SUNDAE
Taken.

>GET FUDGE SUNDAE
```

```
Taken.
```

All's well and good. But it would also maybe be nice to be able to take both at the same time.

```
>GET SUNDAES
fudge sundae:  Taken.
butterscotch sundae:  Taken.
```

That's where the `plural_class` comes in.

```
plural_class sundaes "sundaes"
{
     plural_of fudge_sundae, butterscotch_sundae
     noun "sundaes"
     single_noun "sundae"
}

object fudge_sundae "fudge sundae"
{
     article "a"
     adjective "fudge"
     noun "sundae"
     plural_is sundaes
}

object butterscotch_sundae "butterscotch_sundae"
{
     article "a"
     adjective "butterscotch"
     noun "sundae"
     plural_is sundaes
}
```
[51]

The `plural_of` property on the plural class enumerates the objects which it encompasses; each object encompassed by the plural class then points back to the plural class in its `plural_is` property.

The `plural_verbs` property governs which verbs may be used on the plural object. The `plural_class` class itself provides a default `plural_verbs` which allows basic verbroutines like `DoLook`, `DoDrop`, `DoGet`, and `DoPutIn` to be used. Other actions will result in a response on the order of "You'll have to

[51] One could prevent duplication of properties and other parts of the object definitions by creating a common `sundae` class and deriving both `fudge_sundae` and `butterscotch_sundae` from it, changing only the `adjective` property.

do that one at a time". To change the possible actions for a given plural object, provide a custom `plural_verbs` replacement that returns true only if the `verbroutine` global variable is a valid verbroutine for the object.

Now, consider the following:

```
There are five bananas here.

>GET TWO BANANAS
banana:  Taken.
banana:  Taken.

>INVENTORY
You are carrying two bananas.

>LOOK
There are three bananas here.
```

Something like that can be done easily by creating an identical object from the `identical_class` in the object library. The `identical_class` is similar to the `plural_class` except for a couple details of implementation and behavior.

```
identical_class bananas "bananas"
{
     plural_of banana1, banana2, banana3,
          banana4, banana5
     noun "bananas"
     single_noun "banana"
}

object banana1 "banana"
{
     noun "banana"
     identical_to bananas
}

banana1 banana2 "banana"[52]
{}

banana1 banana3 "banana"
{}

...
```

[52] Using `banana1` as a class to build subsequent banana objects from is a simple way of copying objects (and saves typing and/or copying-and-pasting).

The identical object `bananas` will allow a player to use all the facilities of the `identical_class` in order to manipulate one or more otherwise indistinguishable banana objects.[53]

X.h. Attachables

Ropes and other similar objects—anything, really, which ties onto something else or, even worse, ties *between* two or more objects—are notoriously difficult to code. Safe advice on how to code a rope used to be: code a block of wood instead. The object library provides an `attachable` class which has successfully been used for everything from ropes to blankets to three-ended chains and darts.[54]

Note: Before using attachable objects it is necessary to set the compiler flag `USE_ATTACHABLES`.

The `attachable` class's `attachable_to` property contains a list of all items to which the object may be attached. The `attached_to` property contains a list of all the objects to which the attachable object currently *is* attached. When defined, it must be given an appropriate number of elements. For instance, something that is attachable to only one object would have

```
attached_to 0
```

while, for instance, a rope that can be tied between two other objects must have:

```
attached_to 0, 0
```

The 0 value (the `nothing` object) is just an empty placeholder for the `attached_to` property. If the attachable's initial state is to be attached to a given object, that object can be used instead. For example, a harness that is already attached to a wagon, but which can also be attached to six horses (objects) at the same time, might be initialized as follows:

```
attached_to wagon, 0, 0, 0, 0, 0, 0
```

with room for seven elements.

---

[53] The author encourages the implementation of bananas in any game. More bananas mean more monkeys, and monkeys are always fun.

[54] See **`sample.hug`** for examples of the last two.

The attach_take and attach_drop properties are less frequently used. If attach_take is true, an attempt to take (via calling the DoTake verbroutine) the attachable is made before attaching (or detaching) it. If attach_drop is true, the object is automatically dropped after it is attached.

The attach_verbs and detach_verbs properties contain lists of all valid verbs to attach or detach the object. The DoAttachObject and DoDetachObject verbroutines can be used by all basic attachables, with new grammar specified for the object (corresponding exactly to the verb lists in attach_verbs and detach_verbs) as in:

```
verb "<verb1>"[, "<verb2>",...]
     *                        DoVague
     * object                 DoAttachObject
     * object "to" xobject    DoAttachObject
     ...
```

For instance:

```
verb "tie", "fasten"
     *                        DoVague
     * object                 DoAttachObject
     * object "to" xobject    DoAttachObject
```

DoAttachObject expects a second object (the xobject) to be given as the target for the object to be attached to; the routine itself contains appropriate error-handling if only one object is supplied.

To attach and detach an attachable from an object, use the AttachObject and DetachObject routines:

```
AttachObject(attachable_object, to_object)
```

and

```
DetachObject(attachable_object, from_object)
```

Either routine returns true on success or false on failure.

To check if a particular object is kept immovable by an attachable, call ObjectisAttached(this_object); it returns the object number of the attachable keeping this_object from moving, or false if there is no such impediment and this_object is free to move. Also, any routine that moves the player or player's parent—such as MovePlayer or DoMoveinVehicle—should call MoveAllAttachables to reconcile the location of attached objects (since they are not necessarily connected via the object tree).

Objects with the `mobile` attribute set may be dragged. Non-attachables may have an `attach_immobile` property, which governs whether they may be pulled, dragged, etc. by returning false when freely moveable or true if something is keeping it from moving. In the second case, `attach_immobile` is also responsible for printing any explanatory message.

X.i. What Should I Be Able To Do Now?

By now you should feel comfortable experimenting with the classes in the object library. You should be able to look at the various implementations of scenery, components, characters, doors, vehicles, identical/plural objects, and attachables in existing code (such as in **`sample.hug`**) and not only understand what the various properties of the objects are for, but also how to modify them to achieve a desired effect.

You should, for instance, be able to implement the following:

1. Two `room`s, such as a garden and a shed;
2. A `door` leading into the shed;
3. Various static `scenery` objects in each location;
4. A dozen `identical` roses;
5. A rideable bicycle vehicle kept from going anywhere by a locked attachable bicycle lock; and even
6. A gardener `character` who is capable of answering questions about the things in the shed and the garden.

XI. ADVANCED FEATURES

XI.a. The Display Object

The display object is a special object with which the Hugo Engine interacts to allow the program to be knowledgeable about as well as set certain characteristics of the display. The engine provides access to the following read-only properties (although the names themselves are defined in **`hugolib.h`**):

`screenwidth`	- width of the display, in characters
`screenheight`	- height of the display, in characters
`linelength`	- width of the current text window
`windowlines`	- height of the current text window
`cursor_column`	- horizontal and vertical position of
`cursor_row`	- the cursor in the current text window
`hasgraphics`	- returns true if graphic display is available
`hasvideo`	- returns true if video playback is available
`pointer_x`	- horizontal mouse position (in characters)
`pointer_y`	- vertical mouse position (in characters)

Note: In this usage, "display" refers to the virtual screen usable by the Hugo Engine. Depending on the mode of the engine, this may refer to the full-screen (as for terminal-based ports) or a subsection thereof (i.e., for the engine running in a window).

Additionally, the following display object properties are also writable by a program:

`title_caption`	- sets the window title for the game (where supported)

`needs_repaint`	- set to true when the GUI display changes (such as when window size is changed); may then be reset to false by the program

The Hugo Library also defines the normal read/writable:

`statusline_height`	- of the last-printed status line

In order for the engine to properly identify the display object, it selects the object (if any) with the textual name "(display)", i.e., an object that is defined as

```
object display
{
        ...
}
```

with no explicit textual name. This is how the Hugo Library defines the display object, so that the various display object properties are readable as `display.screenheight`, `display.cursor_column`, etc.

XI.b. Windows

It is possible to create an enclosing window within the full-screen display for text output. Cursor position, line-wrapping, etc. are trimmed to the boundaries of the current window. Cursor positioning and window boundaries are always calculated in fixed-width character dimensions. Various syntaxes for the window statement are:

`window 0`	Restores full-screen output
`window n` `{...}`	Creates a window of `n` lines, bordering on the top edge and sides of the full-screen
`window l, t, r, b` `{...}`	Creates a window with the left-top corner (`l, t`) and the right-bottom corner (`r, b`), where these coordinates are character coordinates on the full-screen
`window` `{...}`	Redraws the last-defined window

Each usage except "window 0" is followed by a block of code during which, normally, text is output to the window. The window (i.e., its boundaries) exists for the duration of the "{...}" block. After it finishes, the top of the main text window is redefined as being immediately below the lowest-drawn window. To clear the record of any window and restore the main text window to the full-screen, use "window 0".

A windowing library file exists called **window.h** which defines a window_class and the associated properties so a window object can be created via:

```
window_class <window name> "title"
{
     win_position    <screen column>, <screen row>
     win_size        <columns>, <rows>

     win_textcolor   <text color>
     win_backcolor   <background color>
     win_titlecolor  <title text>
     win_titleback   <title background>
}
```

The window_class also incorporates the property routines win_init, win_clear, and win_end.

Note: It may be important to keep in mind that measures such as display.screenwidth may change during execution, particularly in a graphical user interface windowing environment which allows resizing of the Hugo program window. For this reason, it is wise to resample display.<property> whenever a window is to be drawn instead of basing the coordinates on, for example, a set of boundaries calculated during program initialization.

XI.c. Reading And Writing Files

There may be times when it will be useful to store data in a file for later recovery. The most basic way of doing this involves

```
x = save
```

and

```
x = restore
```

where save and restore return a true value if successful, or a false value if for some reason they fail. The user is promped for a filename, and, in either case, the entire set of game data—including object locations, variable values, arrays, attributes, etc.—is saved or restored, as appropriate.

Other times, it may be desirable to save only certain values. For example, a particular game may allow a player to create certain player characteristics or other "remembered data" that can be restored in the same game or in different games. To accomplish this, use the writefile and readfile operations.

The structure

```
writefile <filename>
{
      ...
}
```

will, at the start of the writefile block, open <filename> for writing and position <filename> to the start of the (empty) file. (If the file exists, it will be cleared/erased.) At the conclusion of the block, the file will be closed again.

Within a writefile block, write individual values using

```
writeval <value1>[, <value2>, ...]
```

where one or more values can be specified.

To read the file, use the structure

```
readfile <filename>
{
      ...
}
```

which will contain the assignment

```
x = readval
```

for each value to be read, where x can be any storage type such as a variable, property, etc.

For example:

```
local count, test
```

```
count = 10
writefile "testfile"
{
     writeval count, "telephone", 10
     test = FILE_CHECK
     writeval test
}
if test ~= FILE_CHECK    ! an error has occurred
{
     print "An error has occurred."
}
```

will write the variable count, the dictionary entry "telephone", and the value 10 to "testfile". Then,

```
local a, b, c, test

readfile "testfile"
{
     a = readval
     b = readval
     c = readval
     test = readval
}
if test ~= FILE_CHECK    ! an error has occurred
{
     print "Error reading file."
}
```

If the `readfile` block executes successfully, `a` will be equal to the former value `count`, `b` will be "telephone", and `c` will be 10.

The constant `FILE_CHECK`, defined in **`hugolib.h`**, is useful because `writefile` and `readfile` provide no explicit error return to indicate failure. `FILE_CHECK` is a unique two-byte sequence that can be used to test for success. In the `writefile` block, if the block is exited prematurely due to an error, `test` will never be set to `FILE_CHECK`. The `if` statement following the block tests for this. In the `readfile` block, `test` will only be set to `FILE_CHECK` if the sequence of `readval` functions finds the expected number of values in "testfile". If there are too many or too few values in "testfile", or if an error forces an early exit from the `readfile` block, `test` will equal a value other than `FILE_CHECK`.

XI.d. Mouse Input

If the player clicks somewhere on the screen with the mouse pointer (or taps on the screen on a handheld device, or whatever the comparable action is for the platform in question), a Hugo program may read that action. Specifically, a `pause` statement, which normally stores the ASCII value of a keypress in `word[0]`, will instead store the value `MOUSE_CLICK` (defined in **hugolib.h** to be 1) if the mouse is clicked while the engine is waiting for that keypress.

A mouse click (or equivalent action) has no effect during player input—i.e., when the program is waiting for a typed command—unless the individual port provides some behavior such as bringing up a menu, entering a double-clicked word into the input line, etc. The running Hugo program cannot itself monitor mouse input during typed input.

As mentioned above, the display object provides the read-only properties `pointer_x` and `pointer_y`, which give the horizontal and vertical position of the mouse (in characters) respectively.

XII. RESOURCES

XII.a. Creating And Using Resources

The engine allows a Hugo program to access external media data (called resources) compiled into a specially formatted file called a resourcefile. Resourcefiles contain sounds, music, images, and video files used by the program. A resourcefile is created using:

```
resource "<resourcefile>"
{
     "<resource1>"
     "<resource2>"
     ...
}
```

The <resourcefile> name must be 8 or fewer alphanumeric characters which will automatically be converted to all-uppercase. (The reason for this is to maximize portability across different platforms and filenaming systems—unfortunately not everyone adheres to the same conventions, so this restriction is intended to reduce filenaming to the lowest common denominator.)

Currently Hugo supports as resources: JPEG graphic files, RIFF/WAV audio samples, MOD/S3M/XM music modules, MIDI and MP3 songs, and MPEG and AVI movies.[55]

For example, here is an imaginary example resourcefile compiled on a Windows platform:

```
resource "GAMERES1"
{
     "c:\hugo\graphics\logo.jpg"
     "h:\data\scenic panorama.jpg"
     "h:\data\background.jpg"
```

[55] Versions of Hugo prior to v3.0 may not support all resource types. See *APPENDIX F: HUGO VERSIONS* for more information.

```
        "c:\music\intro_theme.s3m"
        "c:\music\theme2.xm"
        "c:\sounds\sample1.wav"
        "c:\sounds\sample2.wav"
}
```

It doesn't matter that the nomenclature within a resource definition is non-portable. In the above "**GAMERES1**", for example, the filenaming is specific to Windows, since that's where the original files will be found. The resources, however, are accessed only by their filenames as entries in the resourcefile index. Therefore, after "**GAMERES1**" is created, the three pictures are referred to as "logo", "scenic panorama" and "background" within the resourcefile "**GAMERES1**". (Note that any drive/path or extension specification is removed and not included in the index. As a result, two resources with the same name but different paths/extensions cannot be written into the same resourcefile.)

Because of the relative non-portability of resourcefiles (plus the additional time it may take on slower machines to index and consolidate potentially hundreds of kilobytes of data), it is recommended that resources be compiled from separate source files than the rest of a Hugo game.

The library extension **`resource.h`** provides useful routines for managing resources in a Hugo program. It also defines the following potential values for the `system_status` global, which may be tested after a resource operation. If `system_status` is non-zero (where zero signifies normal status), it will contain one of the following values[56]:

```
 -1     STAT_UNAVAILABLE
101     STAT_NOFILE
102     STAT_NORESOURCE
103     STAT_LOADERROR
```

XII.b. Pictures

A picture is displayed as a resource in a resourcefile using:

```
picture "<resourcefile>", "<picture>"
```

For example,

```
picture "gameres1", "logo"
```

[56] The result codes are defined in **`resource.h`**.

(It is also possible to enter the path of a picture directly, such as

```
picture "c:\hugo\graphics\logo.jpg"
```

but since this path/filename is obviously operating-system-specific, it should be used for testing only. If the named picture is not found in the given resourcefile, the engine will similarly try to load the picture as an independent file from the current search path(s).)

The picture will be displayed in the currently defined text window. If the picture is smaller than the current window, it will be centered. If larger, it will be shrunk to fit. If the particular version of the Hugo Engine being used is not graphics-enabled, `picture` will have no effect. If the picture is not found or a recoverable error occurs during loading, normal engine execution continues uninterrupted.

resource.h provides a couple of useful routines for managing graphics:

```
LoadPicture("resourcefile", "picture")
LoadPicture("picture")

PictureinText("file", "pic", width, height, preserve)
PictureinText("picture", width, height, preserve)
```

`LoadPicture` is essentially a simple wrapper for the `picture` statement, providing the additional service of checking `display.hasgraphics` to ensure that graphics display is available. `PictureinText` is slightly more complex. It allows a picture to be displayed in the normal flow of text in the main window. The `<width>` and `<height>` arguments give the fixed-width character dimensions of the display area. (Because displays differ in their character dimensions, it is recommended to calculate these based on `display.screenwidth` and `display.screenheight` instead of passing absolute values.) The `<preserve>` parameter, if given, specifies the number of lines (i.e., one or more) at the top of the screen that are protected from scrolling off.

(Either `LoadPicture` or `PictureinText` can be called with only a picture, i.e., with no resourcefile named. In this case, **resource.h** will attempt to find the resource in the last used resourcefile, stored in the `last_resource_file` global. Wherever possible, however, it is recommended to always specify the resourcefile name.)

XII.c. Sound And Music

Sounds and music are played as follows:

```
sound [repeat] <resourcefile>, <resource>[, <vol>]
music [repeat] <resourcefile>, <resource>[, <vol>]
```

The `repeat` keyword is optional; if supplied, it forces the engine to repeatedly play the sound/music resource until further notice (i.e., until it is stopped or a new sound/music resource is played). The `<vol>` argument is optional. If given, it gives a volume percentage (0-100) for playback. Currently playing sound or music can be stopped using:

```
sound 0
music 0
```

`resource.h` provides a pair of wrapper functions to manage playing of audio resources:

```
PlaySound(resourcefile, sample, loop, force)
PlayMusic(resourcefile, song, loop, force)
```

In either case, if `<loop>` is true, it has the same effect as using the repeat token after `sound` or `music`. If `<force>` is true, the sample or song is restarted even if that same sample or song is already playing (otherwise the `PlaySound` or `PlayMusic` call will have essentially no effect). To stop a sample or song from playing via the library interface, use:

```
PlaySound(SOUND_STOP)
PlayMusic(MUSIC_STOP)
```

(where `SOUND_STOP` and `MUSIC_STOP` are constants defined in **`resource.h`**).

XII.d. Video

Video is displayed similarly to static graphic images (that is, it is displayed in the currently window) and controlled similarly to music and sound. The syntax for playing video looks like:

```
video [repeat] <resfile>, <res>[, <vol>, <bkground>]
```

The video resource `res` is played from resourcefile `resfile`, at the volume `vol`. If the optional `repeat` keyword is used, the video plays in a loop, starting over at the beginning when it hits the end. Normally the engine waits for the video to finish playing. If the `bkground` parameter is given and is non-

false, the video plays in the background and the program continues to run, the player may type input, etc. In combination with the `repeat` token, this is useful for creating background/scenic animations.

APPENDIX A: SUMMARY OF KEYWORDS AND COMMANDS

and

Description: Logical AND.

Syntax: `x = <value1> and <value2>`

Result: `x` will be true if `<value1>` and `<value2>` are both non-zero, false if one or both is zero.

anything

Description: Object specifier in grammar syntax line, indicating that any nameable object in the object tree is valid.

array

Description: When used as a data type modifier, specifies that the following value is to be treated as an array address.

Example: `<var1> = array <var2>[<n>]`

The variable `<var2>` will be treated as an array address.

break

Description: Terminates the immediate enclosing loop.

Example:

```
while <expression1>
{
     while <expression2>
```

```
        {
             if <expression3>
                  break
             ...
        }
        ...
}
```

The `break` statement, if encountered, will terminate the innermost loop.

call

Description: Calls a routine indirectly, i.e., when the routine address has been stored in a variable, object property, etc.

Syntax:

```
call <value>[(<arg1>, <arg2>,...)]
```

or

```
x = call <value>(...)
```

where `<value>` is a valid data type holding the routine address.

Value: When used as a function, returns the value returned by the specified routine.

capital

Description: Print statement modifier, indicating that the next word should be printed with the first letter capitalized.

Syntax:

```
print capital <address>
```

where `<address>` is any dictionary word, such as, for example, an `object.name` property.

case

Description: Specifies a conditional case in a `select` structure.

Syntax:

```
select <val>
     case <case1>[, <case2>,...]
          ...
     case <case3>[, <case4>,...]
          ...
```

where `<val>` is value such as a variable, routine return value, object property, array element, etc., and each `<case>` is a single value for comparison (not an expression).

child

Syntax: `x = child(<parent>)`

Return value: The object number of the immediate child object of `<parent>`, or 0 if `<parent>` has no children.

children

Syntax: `x = children(<parent>)`

Return value: The number of child objects possessed by `<parent>`.

cls

Description: Clears the current text window repositions the output coordinates at the bottom left of the text window.

Syntax: `cls`

color (colour)

Description: Sets the display colors for text output.

Syntax: `color <foreground>[, <background>]`

where `<background>` is optional

Parameters: Standard color values for `<foreground>` and `<background>` with constants defined in **`hugolib.h`** (see page 59) are:

0	Black	BLACK
1	Blue	BLUE
2	Green	GREEN
3	Cyan	CYAN
4	Red	RED
5	Magenta	MAGENTA
6	Brown	BROWN
7	White	WHITE
8	Dark gray	DARK_GRAY
9	Light blue	LIGHT_BLUE
10	Light green	LIGHT_GREEN
11	Light cyan	LIGHT_CYAN
12	Light red	LIGHT_RED
13	Light magenta	LIGHT_MAGENTA
14	Light yellow	LIGHT_YELLOW
15	Bright white	BRIGHT_WHITE

dict

Description: Dynamically creates a new dictionary entry at runtime.

Syntax:

```
x = dict(<array>, <maxlen>)

x = dict(parse$, <maxlen>)
```

where `<array>` or `parse$` holds the string to be written into the dictionary, and `<maxlen>` represents the maximum number of characters to be written. Returns the new dictionary address.

Note: Space should be reserved for any dictionary entries to be created at runtime using the `$MAXDICTEXTEND` setting during compilation.

do

Description: Marks the starting point of a `do-while` loop.

Syntax:

```
do
{
      ...
}
while <expr>
```

The loop will continue to run as long as `<expr>` holds true.

elder

Syntax:

```
x = elder(<object>)
```

Return value: The object number of the object preceding `<object>` on the same branch in the object tree. The reverse of `sibling`.

eldest

Same as `child`.

else

Description: In an `if-elseif-else` conditional block, indicates the default operation if no previous condition has been met.

Syntax:

```
if <condition>
      ...
else
      ...
```

elseif

Description: In an `if-elseif-else` conditional block, indicates a condition that will be checked only if no preceding condition has been met.

Syntax:

```
if <condition1>
      ...
elseif <condition2>
      ...
elseif <condition3>
      ...
```

false

Description: A predefined constant value: 0.

for

Description: Loop construction.

Syntax:

```
for (<initial>; <test>; <mod>)
{
      ...
}

for <var> in <object>
{
      ...
}
```

For the first form, where <initial> is the initial assignment expression (e.g. a = 1), <test> is the test expression (e.g. a < 10), and <mod> is the modifying expression (e.g. a = a + 1). The loop will execute as long as <test> holds true.

The second form loops through all the children of <object> (if any), setting <var> to each child object in sequence.

held

Description: Object specifier in grammar syntax line, indicating that any single object possessed by the player object is valid.

hex

Description: Print statement modifier signifying that the following value is not a dictionary address, but should be printed as a hexadecimal number.

Syntax:

```
print hex <var>
```

where, for example, <var> is equal to 26, will print "1A".

if

Description: A conditional expression.

Syntax:

```
if <condition>
        ...
```

where <condition> is an expression or value, will run the following statement block only if <condition> is true.

in

Description: When used in an object definition, places the object in the object tree as a possession of the specified parent. When used in an expression, returns true if the object is in the specified parent.

Syntax:

```
in <parent>
```

or, for example:

```
if <object> [not] in <parent>
{
        ...
}
```

input

Description: Receive input from keyboard, storing the dictionary addresses of the individual words in the word array. Unrecognized words are given a value of 0.

Syntax: `input`

is

Description: Attribute assignment/testing.

Syntax: `<object> is [not] <attribute>`

Usage: When used as an assignment on its own, will set (or clear, if `not` is used) the specified attribute for the given object. May also be used in an expression.

Return value: When used in an expression, returns true if `<object>` has the specified attribute set (or cleared, if `not` is used). Otherwise, it returns false.

jump

Description: Jumps to a specified label.

Syntax: `jump <label>`

where a unique `<label>` exists on a separate line somewhere in the program, in the form:

```
:<label>
```

local

Description: Defines one or more variables local to the current routine.

Syntax: `local <var1>[, <var2>, <var3>,...]`

`locate`

Description: Sets the cursor position within the current text window.

Syntax: `locate(<row>, <column>)`

Note: The maximum horizontal/vertical cursor position is constrained by the boundaries of the current text window. The cursor position is calculated in fixed-width character coordinates.

`move`

Description: Moves an object with all its possessions to a new parent.

Syntax: `move <object> to <new parent>`

`multi`

Description: Object specifier in grammar syntax line, indicating that multiple available objects are valid.

`multiheld`

Description: Object specifier in grammar syntax line, indicating that multiple objects possessed by the player object are valid.

`multinotheld`

Description: Object specifier in grammar syntax line, indicating that multiple objects explicitly not held by the player object are valid.

`music`

Description: Load and play a music resource (if audio output is available).

Syntax:

```
music [repeat] "file", "song"[, vol]
music 0
```

where `<file>` is a compiled Hugo resourcefile, and `<song>` is a music module in MOD, S3M, or XM format. The optional `<vol>` argument, if given, ranges from 0 to 100 and gives a percentage of volume for playback. If the `repeat` token is used, the song continues to loop until either a new song is played, or the current song is stopped (using "`music 0`").

nearby

Description: Used in an object definition to place the object in the specified position in the object tree.

Syntax:

```
nearby <object>
```

Gives the current object the same parent as `<object>`.

```
nearby
```

Gives the current object the same parent as the last-defined object.

newline

Description: Print statement modifier, indicating that a line feed and carriage return should be issued if the current output position is not already at the start of a blank line.

Syntax:

```
print newline
```

not

Description: Logical not.

Syntax:

```
x = not <value>

<object> is not <attribute>
```

Result: In the first example, `x` will be true if `<value>` is false, or false if `<value>` is true.

In the second, the specified attribute will be cleared for `<object>` when used alone as an assignment. As part of an expression, it will return true only if `<object>` does not have `<attribute>` set.

`notheld`

Description: Object specifier in grammar syntax line, indicating that a single object explicitly not held by the player object is valid.

`number`

Description: When used in a grammar syntax line, indicates that a single positive integer number is valid.

When used as a `print` statement modifier, indicates that the following value is not a dictionary address, but should be printed as a positive integer number.

Syntax: (for usage as a `print` statement modifier)

```
print number <val>
```

where, for example, `<val>` is equal to 100, will print "100" instead of the word beginning at the address 100 in the dictionary table.

`object`

Description: Global variable holding the object number of the direct object, if any, specified in the input line.

When used in a grammar syntax line, indicates that a single available object is valid.

or

Description: Logical OR.

Syntax: `x = <value1> or <value2>`

Result: `x` will be true if either `<value1>` or `<value2>` is non-false, or false if both are false.

parent

(Usage 1)

Syntax: `x = parent(<object>)`

Return value: The object number of `<object>`'s parent object.

(Usage 2)

Description: When used in a grammar syntax line, indicates that the domain for validating the availability of the specified direct object should be set to the parent object specified in the input line.

parse$

Description: Read-only engine variable that contains either the offending portion of an invalid input line or any section of the input line enclosed in quotes.

pause

Description: Pauses until a key is pressed. The value of the key is stored in `word[0]`.

picture

Description: Load and display an image resource in the currently defined window (if graphics are available).

Syntax:

```
picture "<resourcefile>", "<picture>"
picture "<picturefile>"
```

where, while `<resourcefile>` is optional, it is very highly recommended (otherwise, `<picturefile>` will likely not be named in a cross-platform portable format).

playback

Description: Plays back recorded commands from a file in place of keyboard input (by prompting the user).

Syntax:

```
x = playback
```

Return value: True if successful, false if not.

print

Description: Print text output.

Syntax:

```
print <output>
```

where `<output>` can consist of both test strings enclosed in quotation marks ("..."), and values representing dictionary addresses, such as object names. Separate components of `<output>` are separated by a semicolon ('`;`'). Each component may also be preceded by a modifier such as `capital`, `hex`, or `number`.

printchar

Description: Prints a character or series of characters at the current cursor position. No newline is printed.

Syntax: `printchar <val1>[, <val2>,...]`

quit

Description: Terminates the game loop.

Syntax: `quit`

random

Description: Engine function which generates a random number.

Syntax: `x = random(<val>)`

Return value: Where `<val>` is a positive integer number, will return a random number between 1 and `<val>`, inclusively.

readfile

Description: A structure that allows values to be read from a file written using `writefile`.

Syntax:

```
readfile <filename>
{
        ...
}
```

The file is opened and positioned to the start at the beginning of the `readfile` block, and closed at the end.

readval

Description: Reads a value in a `readfile` block.

Syntax: `x = readval`

Value: The value read, or 0 in the case of an error. Use the `FILE_CHECK` constant defined in **hugolib.h** to

determine if a `readfile` block has been executed successfully.

`recordoff`

Description: Ends recording commands to a file.

Syntax: `x = recordoff`

Value: True if successful, false if not.

`recordon`

Description: Begins recording commands to a file (by prompting the user).

Syntax: `x = recordon`

Value: True if successful, false if not.

`remove`

Description: Removes an object from the object tree.

Syntax: `remove <object>`

(The same as: `move <object> to 0`)

`restart`

Description: Reloads the initial game data from the .HEX file and calls the `Init` routine.

Syntax: `x = restart`

Note: The `restart` statement does not technically restart the engine; the game loop continues uninterrupted after `Init` is called, only with the game data restored to its initial state.

Value: True if successful, false if not.

`restore`

Description: Restores a saved game's state data from a previously saved file (by prompting the user).

Syntax: `x = restore`

Value: True if successful, false if not.

`return`

Description: Returns from a called routine.

Syntax: `return [<expression>]`

Return value: Returns `<expression>` if provided, otherwise returns false.

`run`

Description: Runs an object property routine if one exists.

Syntax: `run <object>.<property>`

Return value: None; any value returned by the property routine is discarded.

`runevents`

Description: Calls all events which are either global or currently within the event scope of the player object.

Syntax: `runevents`

`save`

Description: Saves the current game state to a file (by prompting the user).

Syntax: `x = save`

Value: True if successful, false if not.

`scriptoff`

Description: Turns transcription off.

Syntax: `x = scriptoff`

Value: True if successful, false if not.

`scripton`

Description: Turns transcription (i.e., recording output to a file or to a printer) on.

Syntax: `x = scripton`

Value: True if successful, false if not.

`select`

Description: Specifies the value for comparison in a `select-case` conditional structure.

Syntax:

```
select <val>
     case <case1>[, <case2>,...]
          ...
     case <case3>[, <case4>,...]
          ...
```

where `<val>` is value such as a variable, routine return value, object property, array element, etc., and each `<case>` is a single value for comparison (not an expression).

serial$

Description: Read-only engine variable that contains the serial number as written by the compiler.

sibling

Syntax: `x = sibling(<object>)`

Return value: The number of the object next to `<object>` on the same branch of the object tree.

sound

Description: Load and play an audio sample resource (if waveform audio output is available).

Syntax: sound `[repeat] "file", "sample"[, vol]`
`sound 0`

where `<file>` is a compiled Hugo resourcefile, and `<sample>` is a waveform sample in RIFF/WAV format. The optional `<vol>` argument, if given, ranges from 0 to 100 and gives a percentage of volume for playback. If the `repeat` token is used, the sample continues to loop until either a new sample is played, or the current sample is stopped (using "`sound 0`").

string

Description: When used in a grammar syntax line, indicates that a string array enclosed in quotation marks is valid.

When used as a function, stores a dictionary entry in a string array.

Syntax: `x = string(<array>, <dict>, <maxlen>)`

`x = string(<array>, parse$, <maxlen>)`

where <array> is an array address, stores the either the dictionary entry given by <dict> or the contents of parse$ as a series of characters, to a maximum of <maxlen> characters. Returns the length of the string stored in <array>.

system

Description: Built-in function to call low-level system functions.

Syntax: `system(<function>)`

FUNC.	LABEL	DESCRIPTION
11	READ_KEY	Read keypress
21	NORMALIZE_RANDOM	Make random values predictable
22	INIT_RANDOM	Restore "random" random values
31	PAUSE_SECOND	Pause for one second
32	PAUSE_100TH_SECOND	Pause for 1/100th of a second
41	GAME_RESET	Returns true after restore or undo
51	SYSTEM_TIME	Stores system time in parse$
61	MINIMAL_INTERFACE	Returns true for minimal ports

(Labels are defined as a constants in **system.h***.)*

If <function> is unavailable, the engine may set system_status to -1 (STAT_UNAVAILABLE).

text

`text to <val>` Sends text to the array table, beginning at address <val>.

`text to 0` Restores normal printing.

to

Description: In a print statement, prints blank spaces in the current background color to the specified position.

Syntax: `print to <val>`

where `<val>` is a positive integer less than or equal to the maximum column position

true

Description: Predefined constant: 1.

undo

Description: Attempts to recover the state of the game data before the last player command.

Syntax: `x = undo`

Value: True if successful, false if not.

verb

Description: Begins definition of a regular verb. Upon returning true from the verb routine, `Main` is called.

Syntax: `verb "<word1>"[, "<word2>",...]`

while

Description: Component of `while` or `do-while` loop construct.

Syntax:

```
while <expr>
{
        ...
}
```

or

```
do
{
        ...
}
while <expr>
```

where the loop will run as long as `<expr>` holds true.

`window`

Description: Switches output to the status window.

Syntax:

```
window a[, b, c, d]
      {...}
```

or

```
window
{...}
```

or

```
window 0
```

If only a single value `<a>` is given, a window of `<a>` lines from the top of the screen is created. If more values are given, a window from top-left (`a`, `b`) to bottom-right (`c`, `d`) is created. If no values are given, the last-defined window is recreated. The new boundaries apply for the length of the following "`{...}`" code block.

"`window 0`" restores full-screen display. There is no following code block.

`writefile`

Description: A structure that writes values to a file that may be read using `readfile`.

Syntax:

```
writefile <filename>
{
      ...
}
```

The file is opened and positioned to the start at the beginning of the `writefile` block, and closed at the end.

writeval

Description: Writes one or more values in a writefile block.

Syntax: writefile value1[, value2, ...]

xobject

Description: Global variable holding the object number of the indirect object, if any, specified in the input line.

When used in a grammar syntax line, indicates that a single available object is valid.

xverb

Description: Begins definition of non-action verb. Upon returning from the verb routine, Main is not called.

Syntax: xverb "<word1>"[, "<word2>", ...]

younger

Same as sibling.

youngest

Syntax: x = youngest(<parent>)

Return value: The number of the object most recently added to parent <parent>.

APPENDIX B: THE HUGO LIBRARY

ATTRIBUTES

`clothing`	for objects that can be worn
`container`	if an object can hold other objects
`enterable`	if an object is enterable
`female`	if a character is female
`hidden`	if an object is not to be listed
`known`	if an object is known to the player
`light`	if an object is or provides light
`living`	if an object is a character
`lockable`	if an object can be locked
`locked`	if an object is locked
`mobile`	if an object can be rolled, etc.
`moved`	if an object has been moved
`open`	if an object is open
`openable`	if an object can be opened
`platform`	if other objects can be placed on it **Note: `container` and `platform` are generally mutually exclusive)**
`plural`	for plural objects (i.e., some hats)
`quiet`	if container or platform is quiet (i.e., the initial listing of contents is suppressed)
`readable`	if an object can be read
`special`	for miscellaneous use
`static`	if an object cannot be taken
`switchable`	if an object can be turned on or off
`switchedon`	if an object is on
`transparent`	if an object is not opaque
`unfriendly`	if a character is unfriendly
`visited`	if a room has been visited
`worn`	if an object is being worn

For system use:

`already_listed`	if object has been pre-listed (i.e., before, for example, a `WhatsIn` listing)
`workflag`	for system use

GLOBALS

The first 12 globals are pre-defined by the compiler:

`object`	direct object of a verb action
`xobject`	indirect object
`self`	self-referential object
`words`	total number of words
`player`	the player object
`actor`	player, or another char. (for scripts)
`location`	location of the player object
`verbroutine`	the verb routine
`endflag`	if not false (0), run `EndGame`
`prompt`	for the player input line
`objects`	the total number of objects
`system_status`	after certain operations

Game setup global variables:

`MAX_SCORE`	total possible score
`MAX_RANK`	up to `x` levels of player ranking
`player_person`	first (1), second (2), or third (3)

Formatting/output global variables:

`FORMAT`	specifies text-printing format
`DEFAULT_FONT`	initially 0; could be set to, e.g., `PROP_ON`
`STATUSTYPE`	0=none, 1=score/turns, 2=time
`TEXTCOLOR`	normal text color
`BGCOLOR`	normal background color
`SL_TEXTCOLOR`	statusline text color
`SL_BGCOLOR`	statusline background color
`INDENT_SIZE`	for paragraph indenting
`AFTER_PERIOD`	string of spaces following a full-stop

Runtime global variables:

`counter`	elapsed turns (or time, as desired)
`event_flag`	set when something happens (see `DoWait`)
`general`	for general use
`light_source`	light source in location
`number_scripts`	number of active character scripts
`obstacle`	if something is stopping the player
`score`	accumulated score
`speaking`	if the player is talking to a character
`verbosity`	for room descriptions
`it_obj`	to reference objects via pronouns
`her_obj`	
`him_obj`	
`them_obj`	

The following are generally for system use, but may be accessed if necessary:

`customerror_flag`	true once `CustomError` is called
`last_object`	set by `Perform` to value of `object`
`list_nest`	used by `ListObjects`
`need_newline`	true when newline should be printed
`old_location`	whenever location changes
`override_indent`	true if no indent should be printed

ARRAYS

`_temp_array[256]`	used by string manipulation functions
`menuitem[11]`	required by the `Menu` function
`oldword[MAX_WORDS]`	for "again" command
`parse_rank[]`	for library parser state
`ranking[]`	in tandem with scoring
`replace_pronoun[]`	for it_obj, him_obj, etc.
`scriptdata[]`	for object scripts
`setscript[]`	the actual scripts

CONSTANTS

`BANNER`	should be printed in every game header

MAX_SCRIPTS	that may be active at one time
MAX_WORDS	in a parsed input line

Color constants:

BLACK	DARK_GRAY
BLUE	LIGHT_BLUE
GREEN	LIGHT_GREEN
CYAN	LIGHT_CYAN
RED	LIGHT_RED
MAGENTA	LIGHT_MAGENTA
BROWN	YELLOW
WHITE	BRIGHT_WHITE
DEF_FOREGROUND	DEF_BACKGROUND
DEF_SL_FOREGROUND	DEF_SL_BACKGROUND
MATCH_FOREGROUND	

Printing format mask constants (for setting the `FORMAT` global):

LIST_F	print itemized lists, not sentences
NORECURSE_F	do not recurse object contents
NOINDENT_F	do not indent listings
DESCFORM_F	alternate room description formatting
GROUPPLURALS_F	list plurals together where possible

Font style mask constants (for use with the `Font` routine):

BOLD_ON	BOLD_OFF	boldface
ITALIC_ON	ITALIC_OFF	italics
UNDERLINE_ON	UNDERLINE_OFF	underline
PROP_ON	PROP_OFF	proportional printing

Additional constants:

UP_ARROW	LEFT_ARROW	for reading keystrokes
DOWN_ARROW	ENTER_KEY	
RIGHT_ARROW	ESCAPE_KEY	
MOUSE_CLICK		

AND_WORD	(“and”)	IN_WORD	(“in”)
ARE_WORD	(“are”)	IS_WORD	(“is”)
HERE_WORD	(“here”)	ON_WORD	(“on”)

`FILE_CHECK`	for verifying `writefile` and `readfile` operations
`MENU_TEXTCOLOR`	normal menu text color
`MENU_BGCOLOR`	normal menu background color
`MENU_SELECTCOLOR`	menu highlight color
`MENU_SELECTBGCOLOR`	menu highlight background color)

PROPERTIES

The first 6 properties are pre-defined by the compiler:

`name`	basic object name
`before`	pre-verb routines
`after`	post-verb routines
`noun (nouns)`	noun(s) for referring to object
`adjective (adjectives)`	adjective(s) describing object
`article`	"a", "an", "the", "some", etc.
`preposition (prep)`	"in", "inside", "outside of", etc.[57]
`capacity`	contains a value representing the capacity of a container or platform
`exclude_from_all`	returns true if the object should be excluded from actions such as ">GET ALL"
`found_in`	in case of multiple virtual (not "physical") parents, `found_in` may hold one or more object numbers; in this case, an "`in <object>`" specifier should probably not be included in the object definition, since `found_in` values are unrelated to "`<object> in <parent>`" relationships

57 Used generally for room objects in order to give a grammatically correct description if necessary; also for containers and platforms.

`holding`	contains a value representing the current encumbrance of a container or platform
`in_scope`	contains a list of actors or objects to which the object is accessible beyond the use of the object tree or the `found_in` property; generally contains either the player object (or, less commonly, another character) and is set using `PutInScope` or cleared using `RemoveFromScope`
`initial_desc`	routine; same as above, but if object has not been `moved` and an `initial_desc` exists, it is called in place of `short_desc`
`list_contents`	a routine that overrides the normal contents listing for a room or object; normal listing is only carried out if it returns false
`long_desc`	routine; detailed description of an object
`misc`	miscellaneous use
`parse_rank`	when there is ambiguity between similarly named objects, the parser will choose the one with a higher `parse_rank` over one with a lower (or non-existent) value; used when `FindObject(<object>, 0)` is called
`pronoun`	"he", "him", "his" or equivalent, so that an object is properly referred to
`reach`	for `enterable` objects such as chairs, vehicles, etc., if the accessibility of objects outside the object in question is limited, `reach` contains a list of the objects which may be accessed

`react_before`	to allow reaction by an object that is not
`react_after`	directly involved in the action
`short_desc`	routine; basic "X is here" description
`size`	for holding/inventory purposes, contains a value representing the size of an individual object
`type`	to identify the type of object, used primarily by class definitions in **`objlib.h`**

For room objects only:

`n_to`
`ne_to`
`e_to`
`se_to`
`s_to`
`sw_to`
`w_to`
`nw_to`
`u_to`
`d_to`
`in_to`
`out_to`
	If a player can move to another location to the north, then `n_to` holds the new room object; if the new object is to the south, `s_to` holds the new object, etc.

`cant_go`	routine; message instead of the default "`You can't go that way.`"

For non-room objects only:

`contains_desc`	a routine that prints the introduction to a list of child objects, instead of the default "`Inside <object> are ...`" or "`<Character> has ...`"; `contains_desc` should always

conclude with a semicolon (';') instead of a new line

`desc_detail` — a routine that prints a parenthetical detail following an object listing, such as: " (which is open)"; the leading space is expected, as are the parentheses, and the `print` statement should conclude with a semicolon (';')

`door_to` — for handling ">ENTER <object>", holds the object number of the object to which an object enters (where the latter behaves as a door or portal)

`ignore_response` — for characters, a routine that runs if the character ignores a player's question, request, etc., instead of the default "X ignores you."

`inv_desc` — a routine that prints a special description when the object is listed as part of the player's inventory; `inv_desc` should conclude with a semicolon (';')

`key_object` — if `lockable`, contains the object number of the key

`order_response` — also for characters, a routine that processes an imperative command addressed to the character by the player; it should return false if no response is provided

`when_open`
`when_closed` — routines; special short descriptions for `openable` objects, where if one exists it is called in place of `short_desc` (when the object is `open` or `not open`, as appropriate) if an `initial_desc` does not exist, or if the object has been `moved`

Note: It is recommended for property routines that print a description—such as `short_desc`, `initial_desc`, etc.—that the routine not simply return true without printing anything as a means of "hiding" the object; such a method may throw text formatting into disarray. The proper means of omitting an object from a list is to set the `hidden` attribute.

For the display object only:

Read-only:

`cursor_column`	horizontal and vertical position of
`cursor_row`	the cursor in the current text window
`hasgraphics`	true if graphics display is available
`hasvideo`	true if video playback is available
`linelength`	width of the current text window
`pointer_x`	fixed-width column of last mouse click
`pointer_y`	fixed-width row of last mouse click
`screenwidth`	width of the display, in characters
`screenheight`	height of the display, in characters
`windowlines`	height of the current text window

Read/writable:

`needs_repaint`	true if the operating system has requested a repaint (for ports which support it)
`statusline_height`	the number of lines used to print the statusline
`title_caption`	dictionary entry giving the full proper name of the program (optional)

While `screenwidth` through `title_caption` are technically defined by **`hugolib.h`** as constants, they are used as property numbers to reference data on the display object.

VERB ROUTINES

The library file **verblib.h** (included by **hugolib.h**) contains a fairly extensive set of basic actions, each of which takes the form Do<verb>, so that the action for taking an object is DoGet, the action for basic player movement is DoGo, etc. Each is called by the engine when a grammar syntax line specifying the particular verb routine is matched. The globals object and xobject are set up by the engine, and the routine is called with no parameters.

Here is a list of the provided verb routines for action verbs:

DoAsk, DoAskQuestion, DoClose, DoDrink, DoDrop, DoEat, DoEmpty, DoEnter, DoExit, DoGet, DoGive, DoGo, DoHello, DoHit, DoInventory, DoListen, DoLock, DoLook, DoLookAround, DoLookIn, DoLookThrough, DoLookUnder, DoMove, DoOpen, DoPutIn, DoPutOnGround, DoShow, DoSit, DoSwitchOff, DoSwitchOn, DoTakeOff, DoTalk, DoTell, DoUnlock, DoVague, DoWait, DoWaitforChar, DoWaitUntil, DoWear

Here are the non-action verb routines:

DoBrief, DoRecordOnOff, DoRestart, DoRestore, DoSave, DoScriptOnOff, DoScore, DoSuperbrief, DoUndo, DoVerbose, DoQuit

Output messages for these verbroutines are handled by the routine VMessage in **verblib.h**.

A set of verb stub routines is also available in **verbstub.h**, including the actions:

DoBurn, DoClimb, DoClimbOut, DoCut, DoDig, DoFollow, DoHelp, DoHelpChar, DoJump, DoKiss, DoNo, DoPull, DoPush, DoSearch, DoSleep, DoSmell, DoSorry, DoSwim, DoThrowAt, DoTie, DoTouch, DoUntie, DoUse, DoWake, DoWakeCharacter, DoWave, DoWaveHands, DoYell, DoYes

The default response for each of these stub routines is a more colorful variation of "Try something else." Any more meaningful response must be incorporated into before property routines. To use these verbs, set the VERBSTUBS flag before compiling **hugolib.h**.

UTILITY ROUTINES, ETC.

First, the junction routines:

`EndGame`	called by the engine via: `EndGame(end_type)` If `end_type` = 1, the game is won; if 2, the game is lost. (Since `endflag` may be any value, a value of, for example, 0 will still call `EndGame`, but with no additional effects via the default `PrintEndGame` routine.) The global `endflag` is cleared upon calling. Returning false from `EndGame` terminates the Hugo Engine. Also calls: `PrintEndGame` and `PrintScore`
`FindObject`	called by the engine via: `FindObject(object, location)` Returns true (1) if the specified object is available in the specified location, or false (0) if it is not. Returns 2 if the object is visible but not physically accessible. The location argument is 0 during object disambiguation performed by the engine. Also calls: `ObjectisKnown`, `ExcludeFromAll`
`Parse`	called by the engine via: `Parse()` Performs all library-side parsing of the player input. Returning true forces the engine to reparse the modified input line. Also calls: `PreParse`, `AssignPronoun` and `SetObjWord`

`ParseError` called by the engine via:
`ParseError(errornumber, object)`

Prints the parsing message/error given by `errornumber`, where an additional object value may also be provided. Returning false signals the engine to print the engine's default error message. Return 2 to force the existing line to be reparsed as is.

May also call: `CustomError`

`Perform` called by the engine via:
`Perform(verbroutine, object, xobject, queue, isxverb)`

Runs the requested verbroutine, setting up the `object` and `xobject` globals if necessary. The `queue` argument is true if more than one call to `Perform` is being made for multiple objects, and the `isxverb` argument is true for verbroutine calls associated with xverb grammar.

`SpeakTo` called by the engine via:
`SpeakTo(character)`

Handles character responses to directed actions. Globals `object`, `xobject`, and `verbroutine` are set up as in a normal verb routine call.

Also calls: `AssignPronoun`

And the routines for grammatically-correct printing:

`Art` calling form:
`Art(object)`

Prints the indefinite article form of the object name, e.g. "`an apple`"

`The`

calling form:
`The(object)`

Prints the definite article form of the object name, e.g. "`the apple`"

`CArt`

calling form:
`CArt(object)`

Prints the capitalized indefinite article form of the object name, e.g. "`An apple`"

`CThe`

calling form:
`CThe(object)`

Prints the capitalized definite article form of the object name, e.g. "`The apple`"

`IsorAre`

calling form:
`IsorAre(object[, formal])`
where the parameter `formal` is optional

Depending on whether or not the specified object is `plural` or singular, prints "`'re`" or "`'s`", respectively (or "` are`" or "` is`" if a non-false `formal` parameter is passed).

`MatchPlural`

calling form:
`MatchPlural(object, w1, w2)`

Prints the dictionary entry given by `w1` if the supplied `object` is not `plural`, or `w2` if it is.

`MatchSubject` calling form:
`MatchSubject(object)`

Matches a verb to the given subject `object`. If the object is `plural`, nothing is printed; if the object is singular, an "`s`" is printed.

Note: None of the above printing routines prints a carriage return, and all return 0 (the empty string). Therefore, either of the following uses are valid:

```
CThe(apple)
print " is here."
```

or

```
print CThe(apple); " is here."
```

Other library routines:

`Acquire` calling form:
`Acquire(parent, object)`

Checks to see if `parent.capacity` is greater than or equal to `parent.holding` plus `object.size`. If so, it moves the object to the specified parent, and returns true. If the object cannot be moved, `Acquire` returns false.

Also calls: `CalculateHolding`

`AnyVerb` calling form:
`AnyVerb(value)`

Returns `value` if the current verbroutine is not an xverb; otherwise it returns false.

`AssignPronoun` calling form:
`AssignPronoun(object)`

Sets the appropriate global `it_obj`, `them_obj`, `him_obj`, or `her_obj` to the specified object.

`CalculateHolding` calling form:
`CalculateHolding(object)`

Properly recalculates `object.holding` based on the sizes of all child objects.

`CenterTitle` calling form:
`CenterTitle(text[, lines])`

Clears the screen and centers the text given by the specified dictionary entry in the top window. The default height of the title (i.e., one line) can be overridden with a second argument giving the number of lines.

`CheckReach` calling form:
`CheckReach(object)`

Checks to see if the specified object is within reach of the player object. Returns true if accessible; returns false and prints an appropriate message if not accessible.

`Contains` calling form:
`Contains(parent, object)`

Returns `object` if the specified object is present as a possession of the specified parent, even as a grandchild[58], otherwise returns false.

[58] A "grandchild" of an object is a child of a child of a given parent object, or a child object thereof, recursively searched.

`CustomError`

calling form:
`CustomError(errornumber, object)`

Replace if custom error messages are desired. Is called by `ParseError` whenever `errornumber` is greater than or equal to 100, specifying a user-provided and user-called parser error. Should return false if no user message is found.

`DarkWarning`

calling form:
`DarkWarning`

Is called by `MovePlayer` whenever the player object is moved into a location without a light source. The default library routine simply prints a message; for a more sinister response or action, such as the demise of the player, replace the default with a new `DarkWarning` routine.

`DeleteWord`

calling form:
`DeleteWord(wordnumber[, number])`

Deletes `number` words—or only one word if no second argument is given—starting with `word[wordnumber]`. Returns the number of words deleted.

`DescribePlace`

calling form:
`DescribePlace(location[, long])`

Prints the location `name` and, when appropriate, a location description (i.e., its `long_desc`). Including a non-false long parameter will always force a location description.

`ExcludeFromAll`

calling form:
`ExcludeFromAll(object)`

Returns true if, based on the current circumstances (verbroutine, etc.), the supplied `object` should be excluded from actions using "`all`"—such as `multi`, `multiheld`, and `multinotheld` grammar tokens.

`FindLight`

calling form:
`FindLight(location)`

Checks to see if a light source is available in `location`; if so, it sets the global `light_source` to the object number of the source and returns that value.

Also calls: `ObjectIsLight`

`Font`

calling form:
`Font(bitmask)`

Sets the current font attributes as specified by `bitmask`, where `bitmask` is one or more font-style constants (see library constants, above) combined with '`|`' or '`+`'.

`GetInput`

calling form:
`GetInput([prompt string])`

Receives input from the keyboard, storing individual words in the `word` array; unknown words—i.e., those that are not in the dictionary—are assigned the empty string, 0 or "". If an argument is passed, it is assumed to be a dictionary address for the `prompt` string. If no argument is passed, no prompt is printed.

`HoursMinutes`

calling form:
`HoursMinutes(counter[, military])`

Prints the time in `hh:mm` format given that `counter` represents the time in minutes from 12:00 a.m. If the optional `military` value is given as a true value, the time is printed in 24-hour format.

`Indent` calling form:
`Indent`

If the `NOINDENT_F` bit is not set in the `FORMAT` mask, `Indent` prints `INDENT_SIZE` spaces without printing a newline.

`InList` calling form:
`InList(object, property, value)`

If the `value` is in the list of values held in `object.property`, returns the element number of the (first) property element equal to `value`; otherwise returns 0.

`InsertWord` calling form:
`InsertWord(wordnumber[, number])`

Makes space for either the number of words given by the `number` argument—or one word if no second `argument` is given—if possible, at `word[wordnumber]`, shifting upward all words from that point to the end of the input line. Returns the number of words inserted.

`IsPossibleXobject` calling form:
`IsPossibleXobject(object)`

Returns true if the object is potentially the xobject in the current command. Does not, however, guarantee that the object *is* an xobject, but is instead a quick and inexpensive utility routine for parsing.

`ListObjects`

calling form:
`ListObjects(object)`

Lists all the possessions of the specified object in the appropriate form (according to the global `FORMAT`). Possessions of possessions are listed recursively if `FORMAT` does not contain the `NORECURSE_F` bit. Format masks are combined, as in:

```
FORMAT = LIST_F | NORECURSE_F | ...
```

Also calls: `WhatsIn`

`Menu`

calling form:
`Menu(number, [width[, selection]])`

Prints a menu, given that the possible `number` of choices (up to 10) are contained in the `menuitem` array, with `menuitem[0]` is the title of the menu. A `width` (in characters) argument and a starting `selection` number are optional. Returns the number of the item selected, or 0 if none is chosen.

Also calls: `CenterTitle`

`Message`

calling form:
`Message(&routine, num, a, b)`

Used by most routines in **hugolib.h** for text output, so that the bulk of the library text is centralized in one location. Message number `num` for the specified routine is printed; `a` and `b` are optional parameters that may represent objects, dictionary entries, or any other value.

(Similar routines are provided in `VMessage` in **verblib.h** and `OMessage` in **objlib.h**.)

`MovePlayer`

calling form:
`MovePlayer(loc[, silent[, none]])`
`MovePlayer(dir[, silent[, none]])`

Moves the player to the new location, properly setting all relevant variables and attributes. If `silent` is passed as a true value, no room description is printed following the move.

A direction object (i.e., `n_obj`, `d_obj`) may be specified instead of a location; in that case, `MovePlayer` moves in that direction from present `location`.

If `none` is true, `before` and `after` routines are not run.

`MovePlayer` can be checked in a location's `before` or `after` property as "`location MovePlayer`" to catch a player's exit from or entrance to a location. In a `before` property, "`object MovePlayer`" can be used to check the target location.

Returns the object number of the new `location`.

`MovePlayer` does not check to see if a move is valid; that must be done before calling the routine.

May also call: `DarkWarning`

`NumberWord`

calling form:
`NumberWord(number[, true])`

Prints a number in non-numerical word format, where `<number>` is between -32768 to 32767. Always returns 0 (the empty string). If a second true argument is supplied, the word is capitalized.

`ObjectIs`

calling form:
`ObjectIs(object)`

Lists certain attributes, such as providing `light` or being `worn`, of the given object in parenthetical form.

`ObjectisKnown`

calling form:
`ObjectisKnown(object)`

Returns true if the object is `known` to the player.

`ObjectisLight`

calling form:
`ObjectisLight(object)`

Returns true if the object or one of its visible possessions is providing `light`. If so, it also sets the global `light_source` the object number of the source.

`ObjWord`

calling form:
`ObjWord(word, object)`

Returns either `adjective` or `noun` (i.e., the property number) if the given is either an adjective or noun of the specified object.

`PreParse`

calling form:
`PreParse`

Provided so that, if needed, this routine may be replaced instead of the more extensive library `Parse` routine. The default `PreParse` routine defined in the library is empty.

`PrintEndGame`

calling form:
`PrintEndGame(end_type)`

Depending on whether `end_type` is 1 or 2, prints, respectively:

"*** YOU'VE WON THE GAME! ***"

or

"*** YOU ARE DEAD ***".

If `end_type` is some other value, nothing is printed.

`PrintScore` calling form:
`PrintScore(end_of_game)`

Prints the `score` in the appropriate form, depending on whether or not `end_of_game` is true.

`PrintStatusLine` calling form:
`PrintStatusLine`

Prints the statusline in the appropriate format, according to the global `STATUSTYPE`.

`PropertyList` calling form:
`PropertyList(object, property)`

Lists the objects held in `object.property` (if any), returning the number of objects listed.

`PutInScope` calling form:
`PutInScope(object, actor)`

Makes the given object accessible to the specified actor, regardless of their respective locations, and providing that the `in_scope` property of the

object has at least one empty slot—i.e., one that equals 0. Returns true if successful.

`RemoveFromScope`

calling form:
`RemoveFromScope(object, actor)`

Removes the given object from the scope of the specified actor. Returns true if successful, or false if the object was never in scope of the actor to begin with.

`SetObjWord`

calling form:
`SetObjWord(position, object)`

Inserts the specified object in the `word` array in the format:

"`adjective1 adjective2 ... noun`"

`ShortDescribe`

calling form:
`ShortDescribe(object)`

Prints the short description (`short_desc`) of the given object, first checking to see if it should run `initial_desc`, `when_open`, or `when_closed`, as appropriate. Then, if no `short_desc` property exists, it prints a default "`X is here.`"

Also calls: `WhatsIn`

`SpecialDesc`

calling form:
`SpecialDesc(object)`

Checks each child object of the specified object, running any appropriate `initial_desc` or `inv_desc` property routines (depending on the calling situation). Sets the global variable

`list_count` to the number of remaining (i.e., non-listed) objects.

`VerbWord` calling form:
`VerbWord`

Returns the dictionary word used as the verb in a typed command.

`WhatsIn` calling form:
`WhatsIn(parent)`

Lists the possessions of the specified parent, according the form given by the global `FORMAT`. Returns the number of objects listed.

Also calls: `SpecialDesc`, `ListObjects`

`YesorNo` calling form:
`YesorNo`

Checks to see if the just-received input is "`yes`", "`y`", "`no`", or "`n`". If none of the above, it prompts for a yes or no answer. Once a valid answer is received, it returns true (if yes) or false (if no).

AUXILIARY MATH ROUTINES:

`abs` calling form:
`abs(a)`

Returns an absolute value given a supplied value.

`higher` calling form:
`higher(a, b)`

Returns the higher number of two supplied values.

`lower`

calling form:
`lower(a, b)`

Returns the lower number of two supplied values.

`mod`

calling form:
`mod(a, b)`

Returns the remainder of one number divided by a second number.

`pow`

calling form:
`pow(a, b)`

Returns one number to the power of another number. (The return value is undefined if the result is outside the boundary of -32768 to 32767.)

STRING ARRAY ROUTINES:

`StringCompare`

calling form:
`StringCompare(array1, array2)`

Returns 1 if `array1` is lexically greater than `array2`, -1 if `array1` is lexically less than `array2`, and 0 if the strings are identical.

`StringCopy`

calling form:
`StringCopy(new, old[, len])`

Copies the contents of the array at the address given by `old` to the array at `new`, to a maximum of `len` characters if `len` is given, or the length of `old` if it isn't.

`StringDictCompare` calling form:
`StringDictCompare(array, dictentry)`

Performs a `StringCompare`-like comparison of a string array given by `array` and the dictionary entry `dictentry`, returning 1, -1, or 0 if `array` is lexically greater than, less than, or equal to `dictentry`, respectively.

`StringEqual` calling form:
`StringEqual(array1, array2)`

Returns true only if `array1` and `array2` are identical.

`StringLength` calling form:
`StringLength(array)`

Returns the length of the string stored as `array`.

`StringPrint` calling form:
`StringPrint(array[, start, end])`

Prints the string stored as `array`, beginning with `start` and ending with `end` if given.

FUSE/DAEMON ROUTINES:

`Activate` calling form:
`Activate(object[, setting])`

Activates the specified `fuse` or `daemon` object. The setting value is only specified for fuses, where it represents the initial value of the timer property.

`Deactivate` calling form:
`Deactivate(object)`

Deactivates the specified `fuse` or `daemon` object.

CHARACTER SCRIPT ROUTINES:

`CancelScript`

calling form:
`CancelScript(character)`

Immediately cancels the character script associated with the object `character`. Returns true if successful, i.e., if a script for `character` is found.

`PauseScript`

calling form:
`PauseScript(character)`

Temporarily pauses the character script associated with the given `character`. Returns true if successful.

`ResumeScript`

calling form:
`ResumeScript(character)`

Resumes execution of a paused script for the given `character`. Returns true if successful.

`SkipScript`

calling form:
`SkipScript(character)`

Skips execution of the script for a given `character` during the next call to `RunScripts` only.

`Script`

calling form:
`Script(character, steps)`

Initializes space for the requested number of `steps` in the `setscript` array, sets up the data for the script in the `scriptdata` array, and returns the location of the script in `setscript`. Returns -1 if `MAX_SCRIPTS` is exceeded.

`RunScripts` calling form:
`RunScripts`

Runs all active scripts, calling them in the form:

```
CharRoutine(character, object)
```

CHARACTER ACTION ROUTINES:

As a starting point, the library also provides a limited number of routines for character scripts to use. They are:

`&CharWait, 0`

`&CharMove, direction_object` (requires **`objlib.h`**)

`&CharGet, object`

`&CharDrop, object`

and

`&LoopScript, 0`

CONDITIONAL COMPILATION:

A number of compiler flags may be set to exclude certain portions of **`hugolib.h`** from compilation if these functions or objects are not required.

FLAG	EXCLUDES
`NO_AUX_MATH`	Auxiliary math routines
`NO_FUSES`	Fuses and daemons
`NO_MENUS`	Use of the `Menu` function
`NO_OBJLIB`	The contents of **`objlib.h`**
`NO_RECORDING`	Command recording functions
`NO_SCRIPTS`	Character scripting routines
`NO_STRING_ARRAYS`	String array functions
`NO_VERBS`	All action verbs
`NO_XVERBS`	All non-action verbs

APPENDIX C: LIMIT SETTINGS

The default settings for the complete set of limits may be obtained by invoking the compiler via:

`hc $list`

The following limits are static and non-modifiable, since they reflect the internal configuration of the Hugo Engine:

`MAXATTRIBUTES`	The maximum number of definable attributes, not counting aliases
`MAXGLOBALS`	The maximum number of definable global variables
`MAXLOCALS`	The maximum number of local variables allowed in a routine, including arguments passed to the routine

The following are the modifiable settings, which may be set using:

```
$<setting>=<new limit>
```

either in the compiler's invocation line or in the source code.

`MAXALIASES`	The maximum number of aliases that may be defined for attributes and/or properties
`MAXARRAYS`	The maximum number of arrays that may be defined (not the total array space, which is automatically reserved)
`MAXCONSTANTS`	The maximum number of constants that may be defined

`MAXDICT`	The maximum number of entries that the compiler can enter into the dictionary table
`MAXDICTEXTEND`	The total number of bytes (not the total number of entries) available for dynamic dictionary extension during runtime
`MAXEVENTS`	The maximum number of global or object-linked events
`MAXFLAGS`	The maximum number of compiler flags that may be set at one time to control conditional compilation
`MAXLABELS`	The maximum number of labels that may be defined during compilation
`MAXOBJECTS`	The maximum number of objects and/or classes that may be created
`MAXPROPERTIES`	The maximum number of properties that may be defined
`MAXROUTINES`	The maximum number of standalone routines (not property routines) that may be defined

APPENDIX D: HUGOFIX AND THE HUGO DEBUGGER

The HugoFix Debugging Library

The HugoFix Debugging Library is a suite of routines that can be used via typed commands in a running Hugo game without the use of any special debugger program. To use HugoFix, set the compiler flag `DEBUG` before including **`hugolib.h`** or any other standard Hugo library files[59]. Then, from the player input line, type:

```
>$?
```

to get a list of all HugoFix debugging commands.

`$?` – Display help menu

Monitoring:

`$fd` – Fuse/daemon monitor on/off
Fuse/daemon monitoring prints verbose information about all starting or stopping fuses or daemons, as well as the value of the `tick` property for any running fuses.

`$fi` – `FindObject` monitoring on/off
`FindObject` monitoring traces calls to the library's `FindObject` routine and their results. This can be extremely useful for debugging things like scope and disambiguation problems.

`$on` – Toggle object numbers
Toggling object numbers on causes an object's numerical value to be displayed after the object name whenever the library functions `The`, `CThe`,

[59] The HugoFix library should only be included during development. As always, when compiling a version for public release, the `DEBUG` flag should be omitted both to keep the filesize of the final Hugo executable down as well as to ensure that debugging functionality is not included in release builds.

`Art`, and `CArt` are called. Can be turned used in conjunction with any other HugoFix function that outputs object names.

$pm – Parser monitoring on/off
Parser monitoring provides information during calls to `Parse`, `ParseError`, and `Perform` (or `SpeakTo`, as applicable) to trace the breakdown, parsing, and execution of a given player input line.

`$sc` – Script monitor on/off
Script monitoring prints verbose information about all starting, stopping, or otherwise running scripts each turn.

Object manipulation:

`$at <obj.> is [not] <attr. #>` – Set or clear object attribute
The object will have attribute number <attr. #> set or cleared. (It's useful to have generated debugging information by passing the –i switch to the compiler in order to get attribute numbers and other useful information.)

`$mo <obj.> to <obj.>` – Move object to new parent
Essentially the same as the Hugo statement: `move <object> to <parent>`. The object will become the youngest child of the parent object.

`$mp <obj.>` – Move player object to new parent
Essentially the same as the Hugo Library function call: `MovePlayer(<obj.>)`. The function may fail (and print an appropriate error message) if the specified parent object is not a valid location (i.e., room or room-equivalent object).

Object testing:

`$fo [obj.]` – Find object (or player, if no object given)
Prints the name of the parent object of a given object (or the player object).

`$na <obj. #>` – Print name of object number
Prints the name of the object specified by object number.

`$nu <obj.>` – Print number of named object
Prints the object number of a given object.

Parser testing:

`$ca` – Check articles for all objects
Useful for preventing forgotten articles in order to avoid something like "`You get apple`" when it should be "`You get the apple`", etc.

`$pc [$all] [obj.]` – Check parser conflicts (for object)
Attempts to determine what objects might be confused with `<obj>` by the parser. May take quite a while if `$all` is specified for a large number of objects.

`$pr` – `parse_rank` monitoring
Monitors how various objects' `parse_rank` property values are evaluated during parsing. Particularly useful with `$fi` and `$pm`.

Other utilities:

`$ac <obj.> [timer]` – Activate fuse (with timer) or daemon
Generally `<obj.>` is an object number, since fuses and daemons are normally not otherwise referred to.

`$de <obj.>` – Deactivate fuse or daemon
Generally `<obj.>` is an object number, since fuses and daemons are normally not otherwise referred to.

`$di [obj.]` – Audit directions (for room object)
Attempts to print out all the possible exits from a given location, or from the present `location` if none is given.

`$kn [<obj. #>]` – Make all object(s) `known`
Sets the `known` attribute for for an object (or for all objects in the game if no single object is specified.

`$nr` – Normalize random numbers
Sets random number generation to predictable values which can be replicated on subsequent playthroughs. Handy for testing things that may be affected by use of the built-in `random` function.

`$ot [obj. | $loc]` – Print object tree (beginning with object)
Prints all the children (beneath a particular object, if given) in tree format.

`$rr` – Restore "random" random numbers
Resets random number generation to produce unpredictable values.

`$uk [<obj. #>]` – Make object unknown
Again for testing involving the `known` attribute. (See `$kn`, above)

`$wo <number>` – Print dictionary word entry `<number>`
Where `<number>` is a value representing a dictionary table address.

`$wn <word>` – Value/dictionary address of (lowercase) word
Where `<word>` is a dictionary entry.

`$au` – Run HugoFixAudit
Runs a number of tests to ensure the validity of certain data, including necessary related properties on individual objects and proper usage of object library classes.

The Hugo Debugger

The Hugo Debugger is a valuable part of the Hugo design system. It allows a programmer to monitor all aspects of program execution, including watching expressions, modifying values, moving objects, etc.—all things expected of a modern source-level debugger.[60]

In order to be used with the debugger, a Hugo program must be compiled using the -d switch in order to create an .HDX debuggable file with additional data such as names for objects, variables, properties, etc.

Note: .HDX files can be run by the engine, but .HEX files cannot be run by the debugger because of the additional data required.

The Unix or MS-DOS convention for running the debugger is:

```
hd <filename>
```

from the command line. In Windows, one may just double-click the debugger's icon to launch it. In either case, the debugger will begin on the debugging screen. Switch back-and-forth from the actual game screen by pressing Tab. At

[60] The Hugo Debugger is not technically a source-level debugger, however. During its development, its author has referred to it as a source(ish) level debugger—what the debugger does, in effect, is to "decompile" compiled code into the tokens and symbols that comprise each line of code. The result is a very close approximation of the original source code.

this point, it is probably best to select "Shortcut Keys" from the Help menu, since the actual keystrokes for running the debugger may vary from system to system. (It is possible to operate the debugger entirely through menus, but this soon becomes tedious for operations like stepping line-by-line.)

The file **`hdhelp.hlp`** should be in the same directory as the debugger program—this is the online help file for the debugger, containing information on such things as:

Printing

Windows and Views, including:

Code Window	Showing the current program exactly as executed, in (almost) source-level format
Watch Window	Allowing any variable expression to be watched/evaluated at any time during execution
Calls	Giving the sequence of nested routine calls at any given point
Breakpoints	Listing all active breakpoints
Local Variables	Listing all local variables, as values, objects, dictionary entries, etc.
Property/Attribute Aliases	
Auxiliary Window	
Output	

Running a program, including:

Finish Routine	While stepping, continues execution without stepping to the end of the current routine
Stepping Through Code	Allows line-by-line execution

Skipping Over Code	Allows the next statement to be passed over without executing
Stepping Backward	Allows retracing of code execution, possibly after values are changed, etc.
Searching Code	Searches the record of executed code for any given string
Watch Expressions	Allows watching multiple variable values or expressions, and to set a breakpoint should a desired value/expression evaluate non-false
Setting or Modifying Values	Any variable, property, array value, or object attribute can be set or reset to a valid value at any point during execution
Breakpoints	A code address, routine, or property routine can be given—control is then passed to the debugger on encountering a breakpoint
Object Tree	At any point, the entire object tree (or just a branch of it) may be displayed
Moving Objects	It is possible to dynamically move objects around the object tree, independent of the program itself
Runtime Warnings	Optional runtime warnings instruct the debugger to alert the user to common causes of problem code which, while syntactically valid and therefore acceptable to the compiler, is in context probably not what was intended.
Setup	Allowing changes (where applicable) in color scheme, printer, etc.

APPENDIX E: PRECOMPILED HEADERS

Note: This section on precompiled headers, while still accurate, becomes less and less vital as computer (and therefore compilation) speeds increase. As of this writing, on a relatively fast computer, a game that takes 6 seconds to compile will compile in 4 using a precompiled version of the library. A game that takes 2 seconds to compile normally will compile in 1. (In other words, the savings are somewhat negligible.)

It is possible to precompile files that would normally be included using the `#include` directive into a precompiled header file that may be linked using `#link`, as in:

```
#link "<filename.hlb>"
```

instead of:

```
#include "<filename.h>"
```

The advantage of doing this is primarily one of faster compilation speed; files that are used over and over again without alteration (such as the Hugo Library) may be precompiled so that they are not recompiled every time.

The `#link` directive must come *after* any grammar, but *before* any new definitions of attributes, properties, globals, objects, synonyms, etc. Grammar is illegal in a precompiled header.

To create a precompiled header, use the -h directive when invoking the Hugo Compiler. The file **hugolib.hug** serves as a good example: it is a small wrapper which compiles the standard Hugo Library. Compile it via

```
hc -h hugolib.hug
```

in order to generate **hugolib.hlb**. Next, change the use of

```
#include "hugolib.h"
```

in a Hugo program to

```
#link "hugolib.hlb"
```

Change the definition for the main routine from

```
routine main
{...
```

to

```
replace main
{...
```

since **hugolib.hug** contains a temporary main routine. The program will now compile (marginally faster) by linking the precompiled library instead of including each uncompiled library file.

Note that any conditional compilation flags set in the Hugo program will have no effect on the compiled code in **hugolib.hlb**, since the routines included in or excluded from **hugolib.hlb** are determined by the flags set in **hugolib.hug**. It is recommended that a Hugo user using precompiled headers compile a version of **hugolib.hug** that includes **hugofix.h** and/or **verbstub.h** as desired.

It is generally not possible to include multiple precompiled headers compiled in separate passes via subsequent `#link`s in the same source file. Because of the absolute references assigned to data such as dictionary addresses, attribute numbers, etc., such an attempt will produce an "incompatible precompiled headers" error.

However, for games that are composed of separate sections that can be combined into distinct files, it may make sense to precompile one **.hug** file containing all the common elements that will be used by the separate sections—such as the player object, etc.—and which `#include`s or `#link`s the library in it. Then, this new **.hlb** file can be `#link`ed in each of the separate sections during development and testing. Of course, each of the separate sections will have to be `#include`d in a single master file for building the full release version.

Finally, it is advisable that precompiled headers be used only in building .HEX files during the design/testing stage in order to facilitate faster development. The reason is that the linker does not selectively include routine calls: the entire **.hlb** file is loaded during the link phase. As a result, Hugo files produced using precompiled headers—especially if existing routines in the **.hlb** file are replaced in the source—tend to be larger and therefore less economical in their memory usage. For this reason, it is recommended that `#include` be used

for building release versions instead of `#link`ing the corresponding precompiled header.

APPENDIX F: HUGO VERSIONS

As of this writing, the latest version of Hugo is 3.1. Most if not all of the actively developed ports are available in v3.1 distributions.

The general rule of thumb is that sequential releases of the Hugo Engine are backward compatible, so that the Hugo Engine in v3.1 is able to run games compiled with v3.0, v2.5, and earlier versions. Earlier versions of the engine, however, are unable to run games compiled with later versions of the compiler. For example, a game compiled with v3.1 cannot be run by the v3.0 engine. (The exceptions to this are v2.5.01 through v2.5.04, which are able to run v3.0-compiled games as a result of the transititional development period for Hugo v3.0. Post-v2.5.04, the official baseline for Hugo releases became v3.0.)

Version 3.1 is syntactically fully compatible with v3.0, the Hugo version which introduced features such as video playback, context menus, and mouse input. 3.1's most notable changes are internal, relating to data storage and code organization, and as such will have little effect on the user. Note however that v2.5 versions of the engine are unable to run v3.1 games.

Here is a quick breakdown of Hugo versions:

VERSION	**SUMMARY**
2.5.0x	Basic Hugo Engine implementation; v3.0-specific language features such as the `video` and `addcontext` keywords unsupported by the v2.5 compiler. Baseline runtime implementation for almost all ports.
3.0	Introduction of additional multimedia and user interface functionality. Additional multimedia including sound, music, and video playback fully supported on Windows, Macintosh, and BeOS.
3.1	Syntactically identical to v3.0; internal format changes.

APPENDIX G: ADDITIONAL RESOURCES

(Please note that while these links were up-to-date as of this writing, the ephemeral nature of the Internet may result in changes, relocations, old sites closing and new sites appearing.)

The Interactive Fiction Archive is the world's number-one repository of publicly available information and tools relating to interactive fiction work and play. It can be found at ***http://www.ifarchive.org***, with a mirror at ***http://mirror.ifarchive.org***.

Two newsgroups serve as the hubs of the interactive fiction community: ***rec.arts.int-fiction***, where the focus is on writing games, and ***rec.games.int-fiction***, which talks about playing them.

The Developers Laboratory at the *Future Boy!* Forum (***http://www.generalcoffee.com/futureboy***) provides a place for Hugo programming discussion.

The Hugonomicon by Cena Mayo at ***http://hugonomicon.sf.net*** provides additional resources for Hugo use and development.

Gilles Duschesne has made available an excellent introduction to Hugo programming in the form of a tutorial available from ***http://www.ifarchive.org/if-archive/programming/hugo/examples/ScavHuntFull.zip***.

A host of general IF-related materials are available at Brass Lantern (***http://www.brasslantern.org***) and PARSIFAL (***http://www.firthworks.com/roger/parsifal/index.html***).

Graham Nelson' s Inform (***http://www.inform-fiction.org***) and TADS, the Text Adventure Design System by Mike Roberts (***http://www.tads.org***), are two other interactive-fiction programming languages.

BOOK 2

TECHNICAL SYSTEM SPECIFICATION

OR

UNDER THE HOOD OF HUGO AND THE .HEX FILE FORMAT

I. INTRODUCTION

Most Hugo programmers will likely never need to bother with the detailed information in this technical guide, but anyone porting Hugo to a new platform, writing an interface or tool for the language, or just interested in taking a closer look at how the Hugo Compiler generates a compiled program (and how the Hugo Engine interprets it) might find a technical specification useful, even if only to verify the occasional behavior or detail. What this look under the hood attempts to do is to outline the configuration of data and code storage used by Hugo, as well as giving an extensive overview of how the various aspects of the language are compiled and interpreted.

This technical specification of the language internals is not a complete programming guide; familiarity with the language and a handy copy of the Hugo Programming Manual will be helpful, as will access to the Hugo source code (written in ANSI C and available at the time of this writing at *ftp://ftp.ifarchive.org/if-archive/programming/hugo/source*).

The standard Hugo source distribution is **`hugov31_source.tar.gz`**. Operating-system-specific sources (i.e., implementations of non-portable functions) are typically **`hugov31_OSname_source.zip`**.

Please note that while this document does address differences between the current version of Hugo and previous versions, it is by no means complete in that respect. For example, a current-version implementation of the Hugo Engine that conforms to this specification is not guaranteed to run programs compiled with all previous versions of Hugo. For further elaboration on such differences, please see the Hugo source itself.

I.a. How Hugo Works

The Hugo system is composed of two parts: the compiler and the engine (the interpreter). (The debugger is actually a modified build of the engine, with an additional command layer to facilitate debugging examination and manipulation of the runtime state.)

The compiler is responsible for reading source files and writing executable code; it does this by first tokenizing a given line of code—breaking it down into a series of byte values representing its contents—and then determining how the

line(s) should be written (i.e., identified, optimized, and encoded) in order to fit properly into the current construct. The compiler is also responsible for organizing and writing tables representing object data, property data, the dictionary, etc.

The engine in turn reads the file produced by the compiler (called a .HEX file, after the default extension), and follows the compiled instructions to execute low-level functions such as object movement, property assignment, text output, and expression evaluation. These low-level operations are, for the most part, transparent to the programmer.

II. ORGANIZATION OF THE .HEX FILE

II.a. Memory Map

If all the separate segments of a .HEX file were stacked contiguously on top of each other, the resulting pile would look something like this:

```
  DATA STORAGE:     MAXIMUM SIZE:
+-----------------+---------------+
| (Link data for  |               |
| .HLB files)     |               |
+-----------------+---------------+
| Text bank       |        16384K |
+-----------------+---------------+
| Dictionary      |           64K |
+-----------------+---------------+
| Special words   |           64K |
+-----------------+---------------+
| Array space     |           64K |
+-----------------+---------------+
| Event table     |           64K |
+-----------------+---------------+
| Property table  |           64K |
+-----------------+---------------+
| Object table    |           64K |
+-----------------+---------------+
| Grammar and     |               |
| Executable code |         1024K |
+-----------------+---------------+
| Header          |      64 bytes |
+-----------------+---------------+
        (Bottom:  $000000)

     MAXIMUM SIZE:  17792K bytes[61]
```

[61] Previously to version 3.1, the size of the grammar and executable code segment was limited to 256K, and the maximum size was 17024K.

Each new segment begins on a boundary divisible by 16; an end-of-segment is padded with zeroes until the next boundary. For each segment, data is general stored in sequential chunks, following two or more bytes giving information about the size of the table.

Dictionary table: the first two bytes give the total number of entries. The third byte is always 0, so that dictionary entry 0 is an empty string (""). Following the dictionary table, a number of bytes may optionally be written for runtime dictionary expansion (where `$MAXDICTEXTEND` is specified at compile-time).

Special words: the first two bytes give the total number of special words.

Array space: the first 480 bytes give the global variable defaults (2 bytes each). For each array entry, the first two bytes give the array length.

Event table: the first two bytes give the total number of events.

Property table: the first two bytes give the total number (*n*) of properties. The following *n**2 bytes give the property defaults.

Object table: the first two bytes give the total number of objects.

II.b. The Header

The header is reserved a total of 64 bytes at the start of the compiled .HEX file, immediately preceding the grammar table. It contains the bulk of information regarding table offsets, junction routine addresses, etc.: essentially, it is a map to where to find things in the file.

Compile with the -u switch to display a map of memory usage in the .HEX file that reflects the offsets and addresses encoded in the header.

Byte	Length	Description
$00	1	Version of Hugo Compiler used[62]
01	2	ID string (compiler-generated)[63]

[62] The version format was changed between v2.0 and v2.1. Version 2.0 programs contained the value 2; version 2.1 programs contain the value 21, version 2.2 programs contain 22, etc.

[63] Pre-v2.3 allowed the programmer to specify an ID string, an unnecessary convention now—the ID string used to be used to create the default savefile name. The ID string is now auto-generated by the compiler and is compared by the engine to the ID of a saved game to see if they match. Precompiled headers have the ID string "$$".

03	8	Serial number
0B	2	Address of start of executable code
0D	2	Object table offset[64]
0F	2	Property table offset
11	2	Event table offset
13	2	Array space offset
15	2	Dictionary offset
17	2	Special words table offset
19	2	`Init` routine indexed address
1B	2	`Main` routine indexed address
1D	2	`Parse` routine indexed address
1F	2	`ParseError` routine indexed address
21	2	`FindObject` routine indexed address
23	2	`EndGame` routine indexed address
25	2	`SpeakTo` routine indexed address
27	2	`Perform` routine indexed address[65]
29	2	Text bank offset

In .HDX (debuggable) Hugo executables only:

3A	1	Debuggable flag, set to 1
3B	3	Absolute start of debugging information
3E	2	Debug workspace (in array table)

A note on data storage: whenever 16-bit words (i.e., two bytes representing a single value) are written or read, it is in low-byte/high-byte order, with the first byte being the remainder of $x/256$ (or the modulus $x\%256$), and the second byte being the integer value $x/256$.[66]

---

[64] Table offsets are equal to the offset of the beginning of the table from the start of data, divided by 16.

[65] Pre-v2.5 had no Perform junction routine; verb routines were called directly by the engine.

[66] For another example, see *APPENDIX A: CODE PATTERNS*. Several of the conditional statements—`if`, `elseif`, etc.—use two bytes to give the absolute skip distance to the next statement if the conditional test fails. The pair is coded in low-byte/high-byte order.

III. TOKENS AND DATA TYPES

The first two places to start inspecting how the Hugo compiler writes a .HEX file are: (1) what byte values are written to represent each individual token (i.e. keywords, built-in functions, etc.), and (2) how different data types and values are formatted.

III.a. Tokens

```
00   (not used)    10   #             20   for
01   (             11   ~             21   return
02   )             12   >=            22   break
03   .             13   <=            23   and
04   :             14   ~=            24   or
05   =             15   &             25   jump
06   -             16   >             26   run
07   +             17   <             27   is
08   *             18   if            28   not
09   /             19   ,             29   true
0A   |             1A   else          2A   false
0B   ;             1B   elseif        2B   local
0C   {             1C   while         2C   verb
0D   }             1D   do            2D   xverb
0E   [             1E   select        2E   held
0F   ]             1F   case          2F   multi

30   multiheld     40   eldest        50   window
31   newline       41   younger       51   random
32   anything      42   elder         52   word
33   print         43   prop#         53   locate
34   number        44   attr#         54   parse$
35   capital       45   var#          55   children
36   text          46   dictentry#    56   in
37   graphics      47   textdata#     57   pause
38   color         48   routine#      58   runevents
39   remove        49   label#        59   arraydata#
3A   move          4A   object#       5A   call
```

```
3B   to            4B   value#         5B   stringdata#
3C   parent        4C   eol#           5C   save
3D   sibling       4D   system         5D   restore
3E   child         4E   notheld        5E   quit
3F   youngest      4F   multinotheld   5F   input

60   serial$       70   readfile
61   cls           71   writeval
62   scripton      72   readval
63   scriptoff     73   playback
64   restart       75   colour
65   hex           76   picture
66   object        77   sound
67   xobject       78   music
68   string        79   repeat
69   array         7A   addcontext[67]
6A   printchar     7B   video[68]
6B   undo
6C   dict
6D   recordon
6E   recordoff
6F   writefile
```

Some of these, particularly the early tokens, are as simple as punctuation marks that are recognized by the engine as delimiting expressions, arguments, etc. Non-punctuation stand-alone tokens (`to`, `in`, `is`) are used for similar purposes, to give form to a particular construction. Others, such as `save`, `undo`, `recordon`, and others are engine functions that, when read, trigger a specific action.

Note also tokens ending with '#': these primarily represent data types that are not directly enterable as part of a program—the '#' character is separated and read as a discrete word in a parsed line of Hugo source. For example, the occurrence of a variable name in the source will be compiled into `var#` (token $45) followed by two bytes giving the number of the variable being referenced. (See the following section on Data Types for more details.)

III.b. Data Types

Internally, all data is stored as 16-bit integers (that may be treated as unsigned as appropriate). The valid range is -32768 to 32767.

---

[67] v3.0 and later
[68] v3.0 and later

Following are the formats for the various data types used by Hugo; to see them in practice, it is recommended to consult the Hugo C source code and the functions `CodeLine()` in **hccode.c**—for writing them in the compiler—and `GetValue()` and `GetVal()` in **heexpr.c**—for reading them via the engine.

ATTRIBUTE:

```
<attr#> <1 byte>
```

The single byte represents the number of the attribute, which may range from $00 to $7F (0 to 127).

Attribute $10, for example, would be written as:

```
$44  10
```

DICTIONARY ENTRY:

```
<dictentry#> <2 bytes>
```

The 2 bytes (one 16-bit word) represent the address of the word in the dictionary table. The empty string ("") is $00.

If the word "apple" was stored at the address $21A0, it would be written as:

```
$46  A0  21
```

OBJECT:

```
<object#> <2 bytes>
```

The two bytes (one 16-bit word) give the object number.

Objects $0002 and $01B0 would be written as, respectively:

```
$4A  02  00
$4A  B0  01
```

PROPERTY:

```
<prop#> <1 byte>
```

The single byte gives the number of the property being referenced.

Property $21 would be written as:

```
$43  21
```

ROUTINE:

```
<routine#> <2 bytes>
```

The two bytes (one 16-bit word) give the indexed address of the routine. All blocks of executable code begin on an address divisible by 16[69]; this allows 1024K of memory to be addressable via the range 0 to 65536. (Code is padded with empty ($00) values to the next address divisible by the address scale.)

For example, a routine beginning at $004004 would be divided by 16 and encoded as the indexed address $0401, in the form:

```
$48  01  04
```

This goes for routines, events, property routines, and even conditional code blocks following `if`, `while`, etc.

VALUE (i.e., INTEGER CONSTANT):

```
<value#> <2 bytes>
```

A value may range from -32768 to 32767; negative numbers follow signed-value 16-bit convention by being $x + 65536$ where x is a negative number.

For example, the values 10 ($0A), 16384 ($4000), and -2 would be written as:

```
$4B  0A  00
$4B  00  40
```

[69] Prior to version 3.1, this scaling factor was 4.

```
$4B  FE  FF     ($FFFE = 65534 = -2 + 65536)
```

VARIABLE:

```
<var#> <1 byte>
```

A program may have up to 240 global variables (numbered 0 to 239), and 16 local variables for the current routine (numbered 240 to 255). Since 240 + 16 = 256, the number of the variable being specified will fit into a single byte.

In the compiler, the first global variable (i.e. variable 0) is predefined as “object”. It would be written as a sequence of two bytes:

```
$45  00
```

A routine’s second argument or local would be numbered 241 (since 240 ($F0) is the first local variable), and would be written as:

```
$45  F1
```

IV. ENGINE PARSING

The engine is responsible for all the low-level parsing of an input line (i.e., player command). Upon receiving an input, the engine parses the line into separate words, storing them in the word array. The word array—i.e., that which is referenced in a Hugo program via `word[n]`—is an internal structure coded using the `word` token instead of `array#`. A static, read-only parser string called `parse$` is used for storage of important data, such as a parser-error-causing word/phrase that cannot otherwise be communicated as an existing dictionary entry.

The first parsing pass also does the following:

1. Allows integer numbers for -32768 to 32767.

2. Time given in "*hh*:*mm*" (hours:minutes) format is converted to an integer number representing the total minutes since midnight, i.e., through the formula: *hh* * 60 + *mm*. The original "*hh*:*mm*" is stored in `parse$`.

3. Up to one word (or set of words) in quotation marks is allowed; if found, it is stored in `parse$`.

4. Special words are processed, i.e., removals and user-defined punctuation are removed, compounds are combined, and synonyms are replaced.[70]

If a user-defined `Parse` routine exists (i.e., if bytes $1D-1E in the header are not $0000), it is called next. If the routine returns true, the engine parsing routine is called a second time to reconcile any changes to the word set.

If at any point the parser is unable to continue, either because an unknown word—one not found in the dictionary table—is found, or because there is a problem later, in grammar matching (described below), a parser error is

[70] See *XI.b Special Words*

generated, and parsing is stopped. (The unknown or otherwise problem-causing word is stored in `parse$`.)

The engine has a set of standard parser errors that may be overridden by a user-provided `ParseError` (i.e., if bytes $1F-20 in the header are not $0000). If there is no `ParseError` routine, or if `ParseError` returns false, the default parser error message is printed.

V. GRAMMAR

The grammar table starts immediately following the header (at $40, or 64 bytes into the .HEX file). It is used for matching against the player's input line to determine the verbroutine to be called, and if applicable, the object(s) and xobject (i.e, the indirect object).

Note: If the input line begins with an object instead of a verb—i.e., if it is directed toward a character, as in "Bob, get the object", then grammar is matched against the phrase immediately following the initial object.)

The grammar table is comprised of a series of verb or xverb (i.e., non-action verb) blocks, each beginning with either `verb` ($2C) or `xverb` ($2D). A $FF value instead of either `verb` or `xverb` indicates the end of the grammar table. A grammar table that looks like

```
000040:  FF
```

has no entries.

Following the verb type indicator is a single byte giving the number of words (i.e., synonyms) for this particular verb. Following that are the dictionary addresses of the individual words.

Think of the simple grammar definition:

```
verb "get", "take"
     * object       DoGet
```

If this were the first verb defined, the start of the grammar table would look like:

```
000040:  2C 02 x2 x1 y2 y1
```

where $*x1x2* is the dictionary address of "get", and $*y1y2* is the dictionary address of "take".

With v2.5 was introduced a separate—although rarely used—variation to the verb header. A verb or xverb definition can contain something like

```
verb get_object
```

where `get_object` is an object or some other value. In this case, the verb word is `get_object.noun` instead of an explicitly defined word. The grammar table in this case would look like”

```
000040:  2C 01 FF FF 4A x2 x1
```

where $FFFF is the signal that instead of a dictionary word address, the engine must read the following discrete value, where $4A is the `object#` token, and $*x1x2* is the object number of `get_object`. This extension is provided so that grammar may be dynamically coded and changed at runtime.

Following the verb header giving the number of verb words and the dictionary address of each is one or more grammar lines, each beginning with a ‘*’ signifying the matched verb word. (For an elaboration of valid grammar syntax specification, please see the Hugo Manual.)

Grammar lines are encoded immediately following the verb header, so that in the first example given above,

```
verb "get", "take"
     * object       DoGet
```

becomes:

```
000040:  2C 02 x2 x1 y2 y1
000046:  08 66 48 r2 r1
00004B:  FF
```

where $r*1r2* is the indexed routine address of DoGet.

The $FF byte marks the end of the current verb definition. Immediately following this is either another `verb` or `xverb` token, or a second $FF to indicate the end of the verb table.

VI. EXECUTABLE CODE

VI.a. A Simple Program

The following is a simple Hugo program:

```
routine main
{
     print "Hello, Sailor!"
     pause
     return
}
```

It will print "Hello, Sailor!", wait for a keypress, and exit. When compiled, the grammar table and executable code look like this:

```
000040:  FF 00 00 00 33 6B 0E 00 5C 79 80 80 83 40 34 67
000050:  75 7D 80 83 86 35 4C 57 21 4C 0D 21 4C 00 00 00
```

Here is what those 32 bytes represent:

```
000040:  FF
```

The grammar table is empty; no grammar has been defined. The first entry in the grammar table is $FF, signifying end-of-table.

```
000041:  00 00 00
```

Padding to the next address boundary.

```
000044:  33
```

A `print` token.

```
000045:  5B 0E 00 5C 79 80 80 83 40 34 67 75 7D 80 83 86 35
                  H  e  l  l  o  ,     S  a  i  l  o  r  !
```

A `stringdata#` ($5B) token of 14 characters ($000E), followed by the encoded string "Hello, Sailor!" (Since this is a `print` statement, the text is written directly into the code instead of in the text bank.)

```
000056:  4C
```

An `eol#` token, to signal end-of-line for the current `print` statement.

```
000057:  57
```

A `pause` token.

```
000058:  21 4C
```

A `return` token, followed by `eol#`. (If there is a value being returned, that expression comes between $21 and $4C. Since in this case the expression is blank—since there is no value being explicitly returned—the $4C comes immediately.)

```
00005A:  0D 21 4C
```

The closing brace symbol $0D marks the end of the routine. All routines are automatically followed by a default $21 and $4C—the equivalent of "`return false`".

VI.b. Expressions

Expressions are encoded as the tokenized representation of the expression. Consider the following code excerpts, assuming that global initializations have included:

```
global glob
array arr[10]
```

and, within the current routine:

```
local loc
```

(Assume also that `glob` and `loc` are the first global variable and first local variable defined.)

1. `loc = 10`

This is coded using the pattern

```
<var#> <1 byte> = <value#> <2 bytes> <eol#>
```

so that the resulting code looks like:

```
45 F0 05 4B 0A 00 4C
loc   =  10
```

The variable number $F0 specifies the first local variable (i.e., local variable 0, where the variable number of local variable *n* is 240+*n*).

2. `glob = 5 * (2 + 1)`

Again, this is coded as a variable assignment:

```
<var#> <1 byte> = <expression> <eol#>

45 0C 05 4B 05 00 08 01 4B 02 00 07 4B 01 00 02 4C
glob  =  5        *  (  2        +  1        )
```

Since the compiler always defines a number of global variables itself, the first-defined global is never 0. If there are 12 pre-defined globals, the first user-defined global has variable number $0C.

3. `arr[loc] = word[2]`

The pattern for this array element assignment is:

```
 <arraydata#> [ <expr> ] = <word> [ <expr> ] <eol#>

 59 F0 00 0E 45 F0 0F 05 52 0E 4B 02 00 0F 4C
 arr      [  loc   ]  = word [ 2        ]
```

(Note that word[n] is not handled the same as array[n].)

4. `array[1] = random(obj.prop #2)`

(Assuming that `obj` and `prop` are the first-defined object and property, respectively.)

```
<arraydata#> [ <expr> ] = random ( <expr> ) <eol#>

59 F0 00 0E 4B 01 00 0F 05 51
arr      [  1        ]  =  random

01 4A 00 00 03 43 06 10 4B 02 00 02 4C
(  obj      .  prop  #  2        )
```

5.

```
glob += (loc++ * arr[7])

45 0C 07 05 01 45 F0 07 07 08
glob  +  =  (  loc   +  +  *

59 F0 00 0E 4B 07 00 0F 02 4C
arr      [  7        ]  )
```

6.

```
if loc = glob + 11
```

(See *APPENDIX A: CODE PATTERNS* for details on how `if` statements and other conditionals are coded.)

```
18 21 00 45 F0 05 45 0C 07 4B 0B 00 4C
if       loc   =  glob  +  11
```

2 bytes give the skip distance (i.e., \$0021 bytes) to the next-executed instruction if the current expression evaluates false.

VII. ENCODING TEXT

Text is written uncompressed into the .HEX file (since there is not really any need for nor any great memory savings from whatever minor compression might be practical). All text, however—including text in `print` statements, dictionary entries, and the text bank—is encoded by adding $14 (decimal 20) to each 8-bit ASCII value in order to prevent casual browsing of game data.

Text in `print` statements is written directly into the code in the form:

```
<stringdata#> <2 bytes> ...encoded string...
```

where the length of the string is given by the first two bytes following <stringdata#>.

Text in dictionary entries is encoded in the dictionary table. A dictionary entry with a given address (*addr*) appears in the dictionary at *addr*+2 (since the first two bytes in the dictionary table are reserved for the number of entries) as:

```
<1 byte> ...encoded dictionary entry...
```

where the maximum allowable length of a dictionary entry is 255 characters.

Text written to the text bank is encoded at a given address in the text bank as:

```
<2 bytes> ...encoded text...
```

where the length of the encoded text is given by the first two bytes. (Note that an address in the text bank requires 3 bytes in the game code, however, since the length of the text bank can exceed 64K.)

VIII. THE OBJECT TABLE

VIII.a. Objects

The object table begins with two bytes giving the total number of objects. The objects then follow in sequential order. Each object requires 24 bytes:[71]

Bytes	
0 - 15	Attributes (128 bits in total, 1 bit/attribute)
16 - 17	Parent
18 - 19	Sibling
20 - 21	Child
22 - 23	Property table position

The offset of any given object n from the start of the object table can therefore be found using:

offset = n * 24 + 2

If a parent has no parent, sibling, and/or child, the appropriate two-byte word is set to $0000.

The property table position represents the offset of the beginning of the given object's property data from the start of the property table, as described below.

VIII.b. Attributes

The 16 bytes of the attribute array contain 8 bits each, giving a total of 128 possible attributes.[72] Essentially, if the bits are thought of sequentially in that the

[71] Pre-v2.1 objects had only 32 possible attributes, and the object size was only 12 bytes, with only 4 bytes given to the attribute array.
[72] In v2.1 and later; there were only 32 attributes in earlier versions

first byte represents attributes 0 to 7, the second byte represents attributes 8 to 15, the third 16 to 23, and the fourth and final byte 24 to 31.

IX. THE PROPERTY TABLE

The property table begins with two bytes giving the total number of properties. This is followed by a list of default property values, each of one 16-bit (2 byte) word each. After this, the properties themselves begin, starting with object 0.

The property values are entered sequentially, with no explicit identification of what object a particular value belongs to. It is the object's object-table entry that gives the location of a given object's property data in the property table.

Each property requires at least 2 bytes:

Byte	
0	Property number
1	Number of data words
2 -	Data in 16-bit word form (2 bytes each)

Property routines are given a "length" of 255 ($FF), which indicates that one word of data follows, representing the (indexed) address of the routine.

At the end of each object in the property table comes the property number 255 ($FF)—not to be confused with the "length" 255, which denotes a routine address. "Property" number 255 is an exception to the two-byte minimum; it does not have any attached length byte or data words. Each object has a place in the object table, even if it has no properties per se. A propertyless object simply has the value 255 at its position in the property table.

(Property data being written for an .HLB linkable file is slightly altered. For example, property routines are marked by $FE instead of $FF. See *XIII.b The Linker*.)

IX.a. Before, After, and Other Complex Properties

Consider the following complex property for an unspecified object:

```
after
{
```

```
        object DoGet
        {
             "You pick up the object."
        }
        object
        {
             "You can't do that with the object."
        }
   }
```

(A simple explanation of the above is that `<object>.after` is called following a call to a verbroutine with which `<object>` was involved. If `<object>` was the object of the verbroutine (i.e., the `object` global), and the `verbroutine` global was `DoGet`, the first block runs. The second block will run if no previous block has run. For a full description of complex properties, see the Hugo Manual.)

First of all, the entry in the property table for `<object>.after` will point to the first line of code in the property routine. Arbitrarily, let's assume this is $000044: the earliest possible code address following a blank grammar table.

```
000040:  FF 00 00 00 45 00 48 1A 00 25 15 00 47 00 00 00
000050:  0D 00 00 00 45 00 25 18 00 47 00 16 00 0D 00 00
000060:  0D 21 29
```

That can be compared to the original source code as:

```
000044:  45 00 48 1A 00
```

The initial "`object DoGet`" block header, assuming that the engine-defined global `object` is global variable number 0, and that the address of `DoGet` is $000068 (represented as an indexed address as $001A).

```
000049:  25 15 00
```

Following the `jump` token ($25) is the indexed address to jump to if "`object DoGet`" isn't matched. In this case, it is $0015, which translates to the absolute address $000054 (i.e., the address of the next header).

```
00004C:  47 00 00 00
```

The <textdata#> label is followed by three bytes giving the address in the text bank of the printed string "You pick up the object."

```
000050:  0D 00 00 00
```

$0D signals the end of this block of executable code, followed by zeroes padding to the next address boundary.

```
000054:  45 00
```

This block header is simply "`object`".

```
000056:  25 18 00
```

As above, following the `jump` token ($25) is the indexed address to jump to if the block header isn't matched. In this case, it is $0018, which translates to $000060 (i.e., the closing $0D of the `after` routine).

```
000059:  47 00 19 00 0D 00 00
```

The second line of text is printed here, followed by $0D to signal the end of this block of code and zero-padding to the next address boundary.

```
000060:  0D 21 29 4C
```

A $0D signals the end of the `after` routine. Property routines are followed by an automatic $21, $29, and $4C (i.e., "`return true`").

X. THE EVENT TABLE

The event table begins with two bytes giving the total number of events. Each event requires 2 bytes:

Bytes	
0 - 1	Associated object (0 for a global event)
2 - 3	Address of event routine

XI. THE DICTIONARY AND SPECIAL WORDS

XI.a. Dictionary

The dictionary begins with two bytes giving the total number of entries. Each dictionary entry is composed of 1 or more bytes:

Byte	
0	Length of entry (number of characters)
1 -	Entry as an encrypted string

XI.b. Special Words

The special words table begins with two bytes giving the total number of entries. Each entry requires 5 bytes:

Byte	
0	Type (0 = synonym, 1 = removal, 2 = compound, 3 = user-defined punctuation)
1 - 2	First dictionary address
3 - 4	Second address (for synonyms and compounds)

XII. RESOURCEFILES

A resourcefile is used to store multiple images, sounds, music tracks, etc. in one manageable file format. The format of a Hugo resourcefile is fairly straightforward.

Every resourcefile starts with a header of 6 bytes:

`00`	'r' **[Note: old 24-bit resourcefiles used 'R']**
`01`	Version number (i.e., 31 for version 3.1)
`02 - 03`	Number of resources
`04 - 05`	Length of index, in bytes

Following the header is the index itself. Each resource entry in the index looks like:

`00`	Length of entry name (i.e., *n* bytes)
`01 - n`	Entry name
`4 bytes`	Offset in resourcefile from end of index
`4 bytes`	Length of resource, in bytes

Note: Older resourcefiles (designated by 'R' in the header) had a limit of 17 MB on resourcefile size (or of any contained resource) and used the following for offset and length:

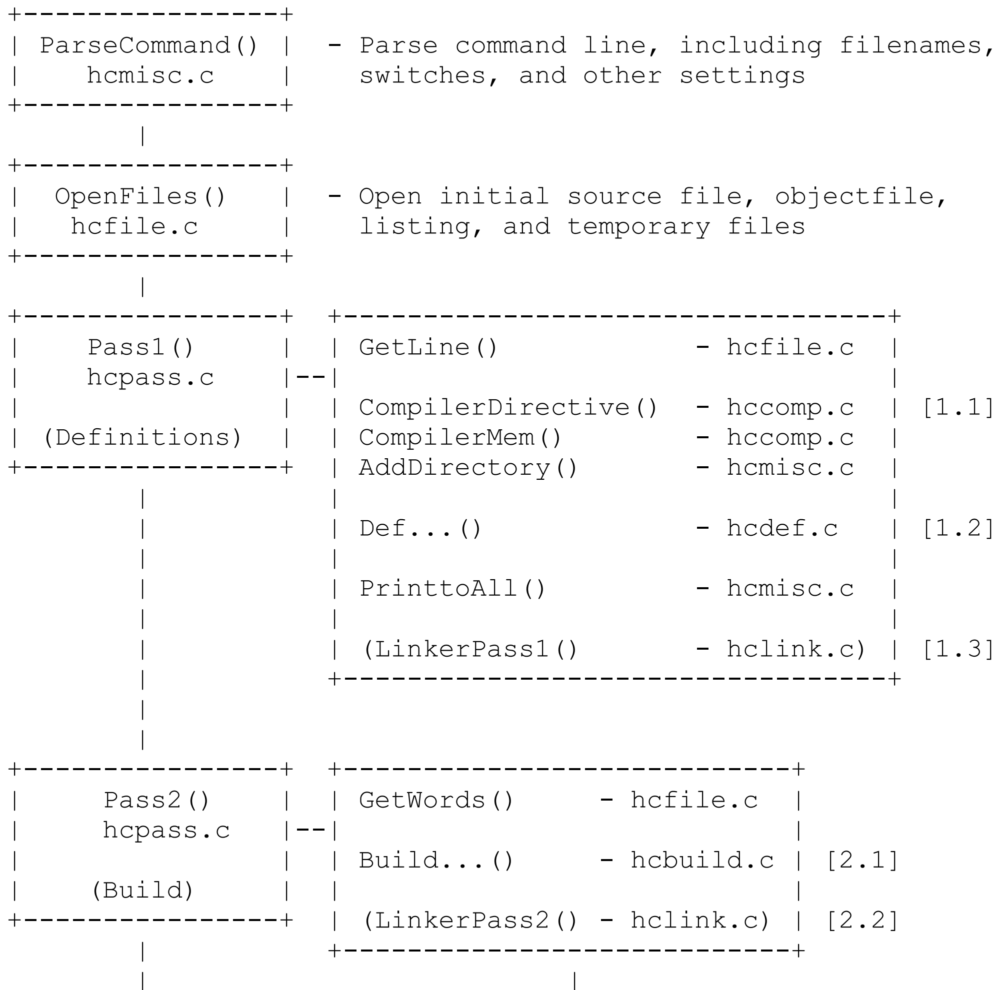
`3 bytes`	Offset in resourcefile from end of index
`3 bytes`	Length of resource, in bytes

These are still supported by the Hugo Engine, but the compiler now writes 32-bit resourcefiles.

Resources are then appended sequentially immediately following the index.

XIII. THE HUGO COMPILER AND HOW IT WORKS

For reference, here is a simplified map of the compiler's function calls, along with the source files in which they are located. The leftmost functions are all called from `main()` in **`hc.c`**:



```
          |                            |
+----------------+        +--------------------------+
|    Pass3()     |        | BuildCode()  - hcbuild.c |  [2.3]
|    hcpass.c    |--+     +--------------------------+
|                |  |                  |
| (Resolve/Link) |  |         +-----------------------+
+----------------+  |         | Code...()  - hccode.c |  [2.4]
                    |         | Codeline() - hccode.c |
                    |         +-----------------------+
                    |                     |          [2.5]
                    |              \+-----------------------+
                    +--------------| Write...()  - hcfile.c |
                                   /| WriteCode() - hcfile.c |
                                    +-----------------------+
```

In **Pass 1**, the initial source file and any included files are read into one contiguous temporary file (called `allfile` in the source). Any compiler directives (i.e., lines beginning with '#', '$' or '@') are processed here [1.1], as are definitions of objects, attributes, properties, global variables, constants, and routines [1.2]. Once a line of source has been parsed and split into discrete words, it is written to `allfile` using `PrinttoAll()`.

Pass 2 is where the bulk of compilation takes place. Lines of pre-parsed source are read from `allfile`. After Pass 1, all symbols (except local variables) are known. Individual constructs such as verbs, objects, routines, and events are processed via `Build...()` functions (i.e., `BuildVerb()`, `BuildObject()`, etc.) [2.1].

At any point in `Pass2()`, the tokenized line currently being processed is held in the global `word[]` array, with the number of tokens in the current line in `words`.

Sections of executable code, such as routines, events, or property routines, are generated by calling `BuildCode()` [2.3], which in turn calls appropriate `Code...()` functions as necessary (i.e., `CodeDo()`, `CodeIf()`, `CodeWhile()`, etc.), or simply `CodeLine()` for any line that doesn't require special treatment [2.4]. Compiled byte-code is emitted to the objectfile via `WriteCode()` [2.5].

(In a departure from the normal order of defining symbols, synonyms, compounds words, removals, and user-defined punctuation are defined in `Pass2()`. Local variables are defined in `BuildCode()`.)

By **Pass 3**, all executable code has been written to the objectfile, structures exist in memory representing to-be-constructed tables, and the text bank (long sections of printed text) exists in a temporary file. First, `ResolveAddr()` (from **`hcmisc.c`**) patches all references that were unknown at the time they were compiled. `Pass3()` then writes the object table, the property table, the event table, the array table, synonyms/removals/compounds/user-defined punctuation, the dictionary, and the text bank.

If a debuggable executable (called an .HDX file) is being generated, the last thing `Pass3()` does is to write the symbolic names of all objects, properties, attributes, aliases, globals, routines, events, and arrays to the end of the file.

XIII.a. Compile-Time Symbol Data

Here are the various structures, arrays, and variables used by the compiler to keep track of symbols at compile-time:

Objects:

`objctr`	total number of objects
`object[n]`	symbolic name of object *n*
`object_hash[n]`	hash value of symbol name
`objattr[n][s]`	attribute set *s* (32 attributes/set)
`oprop[n]`	location in `propdata[]` array
`objpropaddr[n]`	location in property table
`parent[n]`	physical parent
`sibling[n]`	physical sibling
`child[n]`	physical child
`oreplace[n]`	number of times replaced using the `replace` directive

Attributes:

`attrctr`	total number of attributes
`attribute[n]`	symbolic name of attribute *n*
`attribute_hash[n]`	hash value of symbol name

Properties:

`propctr`	total number of properties
`property[n]`	symbolic name of property *n*
`property_hash[n]`	hash value of symbol name
`propset[p]`	true if property *p* has been defined for current object
`propadd[p]`	`ADDITIVE_FLAG` bit is true if property *p* is additive; `COMPLEX_FLAG` bit is true if property *p* is a complex property
`propdata[a][b]`	array of all property data
`propheap`	size of property table

Labels:

`labelctr`	total number of labels
`label[n]`	symbolic name of label *n*
`label_hash[n]`	hash value of symbol name
`laddr[n]`	indexed address of label

Routines:

`routinectr`	total number of routines
`routine[n]`	symbolic name of routine *n*
`routine_hash[n]`	hash value of symbol name
`raddr[n]`	indexed address of routine
`rreplace[n]`	number of times replaced using the `replace` directive

Events (although not really symbols):

`eventctr`	total number of events
`eventin[n]`	object to which event *n* is attached
`eventaddr[n]`	indexed address of event code

Aliases:

`aliasctr`	total number of aliases
`alias[n]`	symbolic name of alias *n*
`alias_hash[n]`	hash value of symbol name
`aliasof[n]`	attribute or property aliased (either the attribute number, or the property number plus `MAXATTRIBUTES`)

Global variables:

`globalctr`	total number of global variables
`global[n]`	symbolic name of global *n*
`global_hash[n]`	hash value of symbol name
`globaldef[n]`	initial value of global at startup

Local variables:

`localctr`	total number of locals defined in the current code block
`local[n]`	symbolic name of local *n*
`local_hash[n]`	hash value of symbol name
`unused[n]`	true until local *n* is used

Constants:

`constctr`	total number of constants
`constant[n]`	symbolic name of constant *n*
`constant_hash[n]`	hash value of symbol name
`constantval[n]`	defined value of constant

Array:

`arrayctr`	total number of arrays
`array[n]`	symbolic name of array *n*
`array_hash[n]`	hash value of symbol name
`arrayaddr[n]`	location in array table
`arraylen[n]`	length of array *n*
`arraysize`	current size of array table

Dictionary:

`dictcount`	total number of dictionary entries
`dicttable`	current size of dictionary
`lexentry[n]`	dictionary entry *n*
`lexaddr[n]`	location of entry n in dictionary table
`lexnext[n]`	location of word following n in the `lexentry[]` array
`lexstart[c]`	location of first word beginning with character *c* in `lexentry[]`
`lexlast[c]`	location of last word beginning with character *c* in `lexentry[]`

Special words:

`syncount`	total number of synonyms, compounds, removals, and user-defined punctuation
`syndata[n]`	`synstruct` structure of *n*

The use of `..._hash[n]` is a rough form of hash-table coding. The compiler, in `FindHash()` in **`hcdef.c`**, produces an *almost* unique value for a given symbol based on the characters in it. Only if `..._hash[n]` matches an expected value does a more expensive `strcmp()` string comparison have to be performed to validate the “match” (or reject it).

XIII.b. The Linker

The compiler has to be able to both create a linkable file (called an .HLB file, as it is usually a precompiled version of the library) and read it back when a `#link` directive is encountered.

In the first case, the compiler writes an .HLB file whenever the -h switch is set at invocation. In order to do that, it does the following things:

1. Property routines, normally marked by a "length" of 255, are changed to a "length" of 254.

2. All addresses are appended to the end of the file instead of being resolved in `Pass3()`. (Labels, being local and therefore not visible outside the .HLB file, are an exception; they are resolved as usual.)

3. Additional data (such as symbolic names) of objects and properties are written in `Pass3()`. Immediately following the object table, the compiler, in `Pass3()`, writes all the relevant data for attributes, aliases, globals, constants, routines.

4. The value "$$" is written into the ID string in the header.

Reading back (i.e., linking) an .HLB file is done in two steps: `LinkerPass1()` [1.3], called from `Pass1()`, and `LinkerPass2()`[2.2], called from `Pass2()`. (The linker routines are found in the source file **`hclink.c`**.)

`LinkerPass1()` simply skims the .HLB file for symbols and defines them accordingly, along with any relevant data. It also reads the .HLB file's text bank and writes it to the current file's temporary file containing the current text bank. Note that since linking must be done before any other definitions, there is no need to calculate offsets here for things like object numbers, addresses in the text bank, etc.

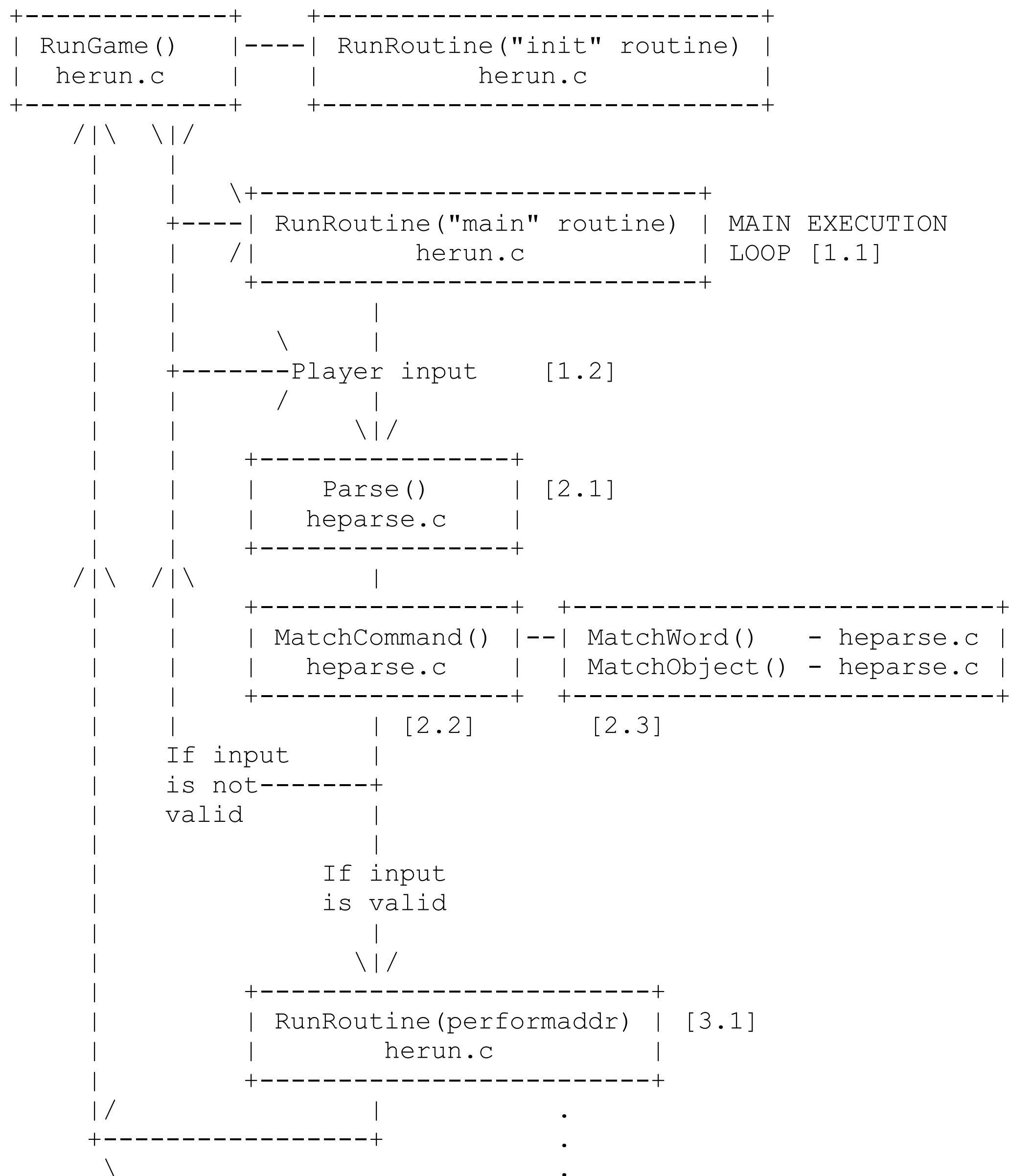
`LinkerPass2()` is responsible for reading the actual executable code. It does this mainly with a simple read/write (in blocks of 16K or smaller). It then reads the resolve table appended to the end of the .HLB file and writes it to the current resolve table so that `Pass3()` can properly resolve the offset code addresses at the end of compilation. (Since the actual start of executable code will vary depending on the length of the grammar table, it is not known at the .HLB file's compile-time what a given address may ultimately be. It is only known that, for example, routine *R* is called from position *P* in the source. Both *R* and *P* must be adjusted for the offset.)

In `Pass3()`, `ResolveAddr()` is now able to resolve addresses from the linked file. Additionally, those properties with a "length" of 254 are adjusted so

that their values—which are really addresses of property routines—are adjusted as per the offset; the "length" of these properties is then written as 255.

XIV. THE HUGO ENGINE AND HOW IT WORKS

Here is a simple map of the main engine loop and the associated functions:



```
                    .
                    .
                    .
         +-----------------------+
         | Expression evaluator: | [4.1]
         |       heexpr.c        |
         |                       |
         | SetupExpr()           |
         |      |                |
         | GetValue()--GetVal()  |
         |      |                |
         | EvalExpr()            |
         +-----------------------+
```

The functions in **herun.c** comprise most of the core game loop and calling points. RunGame() manages the game loop itself [1.1], which can be thought of as being:

Main routine ⇨ Player input ⇨ Parsing ⇨ Action (if valid)

Player input [1.2] is the point at which the engine requests a new input line (usually from the keyboard, but possibly from another source such as a file during command playback).

The **Parsing** section [2.1] refers to the in-engine breakdown and analysis of the input line. The input line is matched against the grammar table in MatchCommand() [2.2]—using MatchWord() and MatchObject() [2.3] to identify either individual words as specified in the grammar, or groups of words that may represent an object name.

If a match is made, the appropriate globals (object, xobject, verbroutine) are set, and Perform() is called [3.1] (or, if Perform() has not been defined, the built-in substitute). (Note that if the command is directed to an object—i.e., another character—SpeakTo() is called instead of Perform().)

RunRoutine() is the method by which any function calls are executed. At any point in RunRoutine() (or in functions called by it), the value mem[codeptr] is the byte value (i.e., the token number) of the current instruction. The value of codeptr advances as execution progresses.

Whenever it is necessary for the engine to evaluate an expression, the expression evaluator subsystem in **heexpr.c** is invoked [4.1]. Here, the eval[] array is initialized with the expression to be evaluated by calling SetupExpr() (which will in turn call GetValue() to sequentially retrieve the elements of the expression). The expression currently in eval[] is solved by calling EvalExpr().

XIV.a. Runtime Symbol Data

Code execution:

`mem[]`	loaded .HEX file image
`defseg`	current memory segment
`codeseg`	code segment (i.e., 0)
`codeptr`	current code position
`stack_depth`	current calling depth

Display:

`pbuffer[]`	print buffer for line-wrapping
`currentpos`	current position (pixel or character)
`currentline`	current row (line)
`full`	counter for `PromptMore()` page-ending
`fcolor`, `bgcolor`, `icolor`, `default_bgcolor`	colors for foreground, background, input, and default background
`currentfont`	current font bitmask
`textto`	if non-zero, text is printed to this array
`SCREENWIDTH`, `SCREENHEIGHT`	maximum possible screen dimensions
`inwindow`	true if in a window
`physical_windowwidth`, `physical_windowheight`, `physical_windowleft`, `physical_windowtop`, `physical_windowright`, `physical_windowbottom`	"physical" window dimensions, in pixels or characters
`charwidth`, `lineheight`, `FIXEDCHARWIDTH`, `FIXEDLINEHEIGHT`, `current_text_x`, `current_text_y`	for font output management

Parsing:

`words`	number of parsed words in input
`word[]`	breakdown of input into words
`wd[]`	breakdown of input into dictionary entries

Arguments and expressions:

`var[]`	global and local variables
`passlocal[]`	locals passed to a routine
`arguments_passed`	number of arguments passed
`ret`	return value (from a routine)
`incdec`	amount a value is being incremented or decremented

Undo management:

`undostack[]`	for saving undo information
`undoptr`	number of operations undoable
`undoturn`	number of operations for this turn
`undoinvalid`	when undo is invalid
`undorecord`	true when recording undo info

XIV.b. Non-Portable Functionality

The Hugo Engine requires a number of non-portable functions which provide the interface layer between the engine and the operating system on which it is running. These functions are:

`hugo_blockalloc`	Large-block `malloc()`
`hugo_blockfree`	Large-block `free()`
`hugo_splitpath`	For splitting/combining filename/path
`hugo_makepath`	elements as per OS naming conventions
`hugo_getfilename`	Asks the user for a filename
`hugo_overwrite`	Verifies overwrite of a filename
`hugo_closefiles`	`fcloseall()` or equivalent
`hugo_getkey`	`getch()` or equivalent
`hugo_getline`	Keyboard line input
`hugo_waitforkey`	Cycles while waiting for a keypress
`hugo_iskeywaiting`	Reports if a keypress is waiting
`hugo_timewait`	Waits for 1/*n* seconds
`hugo_init_screen`	Performs necessary display setup
`hugo_hasgraphics`	Returns graphics availability
`hugo_setgametitle`	Sets title of window/screen

`hugo_cleanup_screen`	Performs necessary screen cleanup
`hugo_clearfullscreen`	Clears entire display area
`hugo_clearwindow`	Clears currently defined window
`hugo_settextmode`	Performs necessary text setup
`hugo_settextwindow`	Defines window in display area
`hugo_settextpos`	Sets cursor/text-output position
`hugo_scrollwindowup`	Scrolls currently defined window
`hugo_font`	Sets font for text output
`hugo_settextcolor`	Sets foreground color for text
`hugo_setbackcolor`	Sets background color for text
`hugo_color`	Returns a valid color reference
`hugo_print`	Outputs formatted text
`hugo_charwidth`	Returns width of a given character
`hugo_textwidth`	Returns width of a given string
`hugo_strlen`	`strlen()` for embedded codes
`hugo_specialchar`	Translation for special characters
`hugo_hasvideo`	Returns video availability[73]

For elaboration of the intent and implementation of these functions, see **`heblank.c`** in the standard source distribution (**`hugov31_source.tar.gz`**), or one of the implementations such as **`hemsvc.c`** (in **`hugov31_win32_source.zip`**, the Windows source package), **`hegcc.c`** (in **`hugov31_unix_source.tar.gz`**, the gcc/Unix package), etc.

XIV.c. Savefile Format

Hugo saves the game state by (among other things) saving the dynamic memory from start of the object table to the start of the text bank (i.e., including objects, properties, array data, and the dictionary). It does this, however, in a format that only notes if the data has changed from its initial state.

The structure of a Hugo savefile looks like this:

`0000 - 0001`	ID (assigned by compiler at compile-time)
`0002 - 0009`	Serial number
`000A - 0209`	All variables (global and local, 256*2 bytes)

[73] v3.0 and later

020A -	Object table to text bank (see below)
n bytes	Undo data (where n = `MAXUNDO`*5*2 bytes)
2 bytes	undoptr
2 bytes	undoturn
1 byte	undoinvalid
1 byte	undorecord

In saving from the object table up to the start of the text bank, the engine performs a comparison of the original gamefile against in-memory dynamic data (which may have changed).

If a given byte *n* in a savefile is non-zero, it represents that the next *n* sequential bytes are identical between the gamefile and the saved data. If *n* is 0, the byte *n*+1 gives the value from the memory image. (Although it takes 2 bytes to represent a single changed byte, the position within both the gamefile and the memory image only increases by 1.)

The practical implementation of the Hugo savefile format is found in `RunSave()` and `RunRestore()` in **`herun.c`**.

XV. DARK SECRETS OF THE HUGO DEBUGGER

The Hugo Debugger is basically a modified build of the Hugo Engine; the two share the same core code for program execution, but the debugger wraps it in a calling framework that allows the user (or the debugger itself) to control—i.e., start, stop, or step through—execution.

The key difference with the debugger build of the engine is in `RunRoutine()`, which in the debugger looks more like this:

```
          ...
           |
           |
    +---------------+     +------------+
    | RunRoutine() |---->| Debugger() |  (if debugger_interrupt
    |   herun.c    |     |   hd.c     |   is non-false)
    +---------------+     +------------+
           |
           |
          ...
```

The debugger build contains a global flag called `debugger_interrupt`; if this flag is non-false, `RunRoutine()` is interrupted before executing the next instruction.

The `Debugger()` function is responsible for switching to and updating the debugger display. `Debugger()` is also the hub for any debugger functions initiated by the user, such as setting breakpoints, setting watch expressions, changing values, moving objects, etc.

The debugger controls program execution by returning from `Debugger()` to `RunRoutine()`. If `debugger_interrupt` is true, only the current instruction will execute, then control will pass back to `Debugger()` (i.e., stepping). In order to resume free execution, `Debugger()` returns with `debugger_interrupt` set to false.

A number of other variables in the debugger influence program execution in addition to `debugger_interrupt`:

`debugger_run`	true when engine is running freely
`debugger_collapsing`	true when collapsing the call

`debugger_step_over`	true if stepping over (i.e., same-level stepping)
`debugger_skip`	true if skipping next instruction
`debugger_finish`	true if finishing current routine
`debugger_step_back`	true if stepping backward
`step_nest`	for stepping over nested calls (i.e., with `debugger_step_over`)

XV.a. Debugger Expression Evaluation

The debugger must evaluate expressions in several contexts, including when solving watch expressions and when changing an existing value. (In-debugger expression management is contained primarily in **`hdval.c`**.)

In order to do this, the debugger includes a minimal version of the compiler's expression parser. It parses a user-supplied expression in the function `ParseExpression()`. What `ParseExpression()` does is to essentially compile that expression, storing the result in the debug workspace in the array table. (Remember that the address of the debug workspace—256 bytes after any user-defined array storage—is found in the header in .HDX files.)

After writing the expression, the debugger can then set `codeptr` to the start of the debug workspace, then call the engine's `SetupExpr()` and `EvalExpr()` functions as it would to evaluate any other expression.

XV.b. The .HDX File Format

The .HDX file format for Hugo debuggable executables, as well as having some additional information in the header (see *II.b The Header*) and a 256 byte workspace reserved at the end of the array table, appends symbolic debugging data as follows:

Object names	For each object: 1 byte giving the length, followed by the name as a string
# of properties	2 bytes
Property names	For each property: 1 byte (length), then the name
# of attributes	2 bytes
Attribute names	For each attribute: 1 byte (length), then the name

# of aliases	2 bytes
Alias names	For each alias: 1 byte (length), then the name, then two bytes for the association
# of routines	2 bytes
Routine names	For each routine: 1 byte (length), then the name
# of events	2 bytes
Event data	4 bytes for each—2 bytes for the parent; 2 bytes for the address
# of arrays	2 bytes
Array data	For each array: 1 byte for the name length, followed by the name, followed by 2 bytes for the address

(Note that it isn't necessary to store the total number of objects, since that is already available at the start of the normal object table.)

APPENDIX A: CODE PATTERNS

What follows is a detailed breakdown of how the set of valid tokens in Hugo is encoded and read within compiled code.

Tokens simply marked **`TOKEN`** are coded just as the byte value of the token in question; no other formatting or necessary token/value is required to follow. These are typically used for delimitation, signaling the end of a structure or structure component, etc.

`STATEMENTS` are those tokens that are read by the engine as some sort of operation—typically, these are "start of line" tokens, with some exceptions.

`VALUES` return an integer value to the engine within the context of an expression. See *III.b Data Types*, which describes all the valid types of values.

`INTERNAL` tokens never appear in source code. These are added by the compiler for use by the engine.

A "code block" is any executable statement or statements followed by a terminating $0D ('}').

Constructions may include *expressions* or *values*; the difference between the two is that values are expected to be discrete data types. Note also that `GetVal()` in **`heexpr.c`** allows a solvable expression bracketed by $01 ('(') and $02 (')') to be treated as a discrete value.

Source references point to places in the Hugo C source code that may help to clarify how a particular construction is coded/interpreted. While not specifically mentioned, the compiling of many tokens is localized in `CodeLine()` in **`hccode.c`**, and the execution of many simple statements is localized in `RunRoutine()` in **`herun.c`**. The reading of values from data types or expressions begins with `GetValue()` in **`heexpr.c`**, with the basic identification of values in `GetVal()`.

`01 ( TOKEN`

`02 ) TOKEN`

`03 . TOKEN`

`04 : reserved (not coded)`

```
05    =        TOKEN

06    -        TOKEN

07    +        TOKEN

08    *        TOKEN

09    /        TOKEN

0A    |        TOKEN

0B    ;        TOKEN

0C    {        TOKEN

0D    }        TOKEN
```

(Signifies the end of a code block)

```
0E    [        TOKEN

0F    ]        TOKEN

10    #        TOKEN

11    ~        TOKEN

12    >=       TOKEN

13    <=       TOKEN

14    ~=       TOKEN

15    &        TOKEN

16    >        TOKEN

17    <        TOKEN
```

18 if STATEMENT

```
18 <skip distance> <expression> 4C
     <conditional block>
<next statement>
```

As in:

```
if <expression>
{...}
```

Where the two bytes of `<skip distance>` are the absolute distance—in low-byte/high-byte order—from the first byte of the pair to the next line of code that will execute if `<expression>` evaluates to false, i.e., the distance to `<next statement>`. If `<expression>` evaluates to a non-false value, `<conditional block>` is run. Note that $4C indicates end-of-line.

`<expression>` is simply a tokenized representation of the expression as it appears in the source line.

Source: **hccode.c** – `CodeIf()`
herun.c – `RunIf()`

19 , TOKEN

1A else STATEMENT

```
1A <skip distance>
     <conditional block>
<next statement>
```

As in:

```
else
{...}
```

Where `<conditional block>` runs only if no immediately preceding `if` or `elseif` condition has been met. If a previous condition has been met, control passes ahead to `<next statment>`, i.e., forward the number of bytes given by the two bytes of `<skip distance>`.

Source: **hccode.c** – `CodeLine()`

herun.c – RunIf()

1B elseif STATEMENT

```
1B <skip distance> <expression> 4C
     <conditional block>
<next statement>
```

As in:

```
elseif <expression>
{...}
```

See if.

Source: **hccode.c** – CodeIf()
herun.c – RunIf()

1C while STATEMENT

```
:<starting point>
1C <skip distance> <expression> 4C
     <conditional block>
25 <starting point>
<next statement>
```

As in:

```
while <expression>
{...}
```

As long as <expression> evaluates to a non-false value, <conditional block> is run. Note the implicit jump ($25) coded by the compiler to maintain the loop—<starting point> is only an address; only the two-byte address following $25 is written as a jump-back point. See if.

Note that because the <starting point> is written as a two-byte indexed address, it must begin on an address boundary, padded with empty ($00) values, if necessary.

Source: **hccode.c** – CodeWhile()
herun.c – RunIf()

1D do STATEMENT

```
1D <skip distance>
:<starting point>
     <block>
1C <two bytes> <expression> 4C
<next statement>
```

As in:

```
do
{...}
while <expression>
```

If, after `<block>` executes, `<expression>` evaluates to a non-false value, the engine returns to `<starting point>` (which must begin on an address boundary). The two bytes following `while` ($1C) match the syntax of the normal `while` loop, but are undefined for this usage. Instead, the distance to the next statement is given after the `do` token ($1D) in the two bytes of `<skip distance>`.

Source: **hccode.c** – `CodeDo()`
herun.c – `RunDo()`

1E select STATEMENT

```
1E
```

When encountered by the engine, resets the conditional-statement evaluator, i.e., so that the next `case` conditional is treated as an `if` instead of an `elseif`. Note that the variable that follows `select` in a line of source code is not coded here (but it is needed by the compiler to construct subsequent `case` statements).

See `case`.

Source: **hccode.c** – `CodeSelect()`
herun.c – `RunIf()`

1F case STATEMENT

Treated identically by the engine to `elseif` once a `select` token ($1E) has reset the conditional-statement evaluator to no previous matches.

In other words, what the compiler does is take:

```
select <expression>
     case <test1>
          <first conditional block>
     case <test2>
          <second conditional block>
     ...
     case else
          <default conditional block>
```

and restructure it into:

```
1F <skip distance> <expression> 05 <test1> 4C
     <first conditional block>
1F <skip distance> <expression> 05 <test2> 4C
     <second conditional block>
1A <skip distance>
     <default conditional block>
```

Note that $1A is the `else` token, $05 is the '=' token, and that the two bytes of `<skip distance>` give the distance to the next `case`.

Source: **hccode.c** – `CodeSelect()`
herun.c – `RunIf()`

20 for STATEMENT

```
<assignment>
:<starting point>
20 <skip distance> <expression> 4C
     <conditional block>
     <modifying expression>
     25 <starting point>
<next statement>
```

As in: `for (<assign>; <expr>; <modifying>)`

```
{...}
```

The `<assignment>`, if given in the source code, is coded as a regular executable assignment of some data type. Again, nothing is explicitly coded at `<starting point>`—it is simply a reference point for the `jump` ($25) to return to. The `for` ($20) line operates as a regular conditional test (see `if`). The `<modifying expression>` is appended after the conditional block is coded. This, like the `<assignment>` is simply a regular executable assignment.

Source: **hccode.c** – `CodeFor()`
herun.c – `RunIf()`

21 return STATEMENT

```
21 <expression> 4C
```

As in:

```
return <expression>
```

Where `<expression>` is optional, so that a standalone `return` order can be coded as:

```
21 4C
```

22 break STATEMENT

```
22
```

23 and TOKEN

24 or TOKEN

25 jump STATEMENT

```
25 <address>
```

As in:

```
jump <label>
```

Where `<address>` is two bytes giving the indexed address of the next statement to be executed. (The `<label>` is coded as `<address>`.)

26 run STATEMENT

`26 <value> 4C`[74]

Where `<value>` is simply read and forgotten, as in running an `object.property` property routine and throwing away the value.

27 is TOKEN

As in: `<object> is <attribute>` (statement form)
`<object> is <attribute>` (value form).

28 not TOKEN

29 true VALUE

`29`

Hard-coded Boolean constant meaning 1.

2A false VALUE

2A

Hard-coded Boolean constant meaning 0.

2B local reserved (not coded)

2C verb STATEMENT

`2C <n> <dict_1> <dict_2>...<dict_n>`

[74] Pre-v2.3 omitted the `eol#` marker ($4C).

Occurs in the grammar table and explicitly denotes the beginning of a new verb, where the single byte `<n>` gives the number of dictionary words coded immediately following representing synonyms for this verb.

2D xverb STATEMENT

```
2D <n> <dict_1> <dict_2>...<dict_n>
```

Coded and handled identically to `verb`, except that it is flagged differently so the engine knows it is a “non-action”.

2E held GRAMMAR TOKEN

2F multi GRAMMAR TOKEN

30 multiheld GRAMMAR TOKEN

31 newline PRINT TOKEN

Signals a `print` statement to issue a newline *only* if one is needed.

32 anything GRAMMAR TOKEN

33 print STATEMENT

```
33 <print data> 4C
```

```
33 <print data> 0B <print data> ... 4C
```

Where <print data> is one of the following:

```
stringdata#
```

any value, treated as a dictionary entry

```
parse$
serial$

newline
capital
number
hex
```

Multiple `<print data>` sequences are separated by a semicolon (';') token ($0B).

Source: **`herun.c`** – `RunPrint()`

34 number GRAMMAR TOKEN or PRINT TOKEN

In a `print` statement, signals that the following value should be printed as a number, not as the corresponding dictionary entry.

In a grammar line, represents any integer number.

35 capital PRINT TOKEN

Signals that the following dictionary entry should have its first letter capitalized.

36 text STATEMENT

`36 3B <value> 4C`[75]

As in: `text to n`

Where <value> is either an address in the array table, or constant 0 (to restore text output to the standard display).

[75] Pre-v2.3 omitted the `eol#` marker ($4C).

37 graphics STATEMENT

(Not implemented.)

38 color STATEMENT

```
38 <value> 4C
38 <value> 19 <value> 4C
38 <value> 19 <value> 19 <value> 4C
```

As in:

```
color foreground
color foreground, background
color foreground, background, inputcolor
```

Where `<value>` is a Hugo color value from 0 to 17 giving the foreground text color. If a second value is given, separated by a comma ($19), it represents the background color. If a third value is given, separated by a comma ($19), it represents the input color.

39 remove STATEMENT

`39 <value> 4C`[76]

As in: `remove <object>`

Source: **herun.c** – `RunMove()`

3A move STATEMENT

`3A <value> 3B <value> 4C`[77]

As in: `move <object1> to <object2>`

Source: **herun.c** – `RunMove()`

3B to TOKEN

Followed by a value, as in:

---

[76] Pre-v2.3 omitted the `eol#` marker ($4C).
[77] Pre-v2.3 omitted the `eol#` marker ($4C).

```
3B <value>
```

Typically found in “`print to n`”, “`text to n`”, etc., in which case the line will finish with `eol#`:

```
...3B <value> 4C
```

3C **`parent`** **VALUE**

```
3C 01 <expression> 02
```

As in: `parent(...)`

Returns the parent object of the object resulting from `<expression>`.

(Alternate usage is as a grammar token, coded simply as $3C with no following parenthetical expression.)

3D **`sibling`** **VALUE**

```
3D 01 <expression> 02
```

As in: `sibling(...)`

Returns the sibling of the object resulting from `<expression>`.

3E **`child`** **VALUE**

```
3E 01 <expression> 02
```

As in: `child(...)`

Returns the child object of the object resulting from `<expression>`.

3F **`youngest`** **VALUE**

```
3F 01 <expression> 02
```

As in: `youngest(...)`

Returns the youngest (most recently added) child object of the object resulting from `<expression>`.

40 eldest VALUE

`40 01 <expression> 02`

As in: `eldest(...)`

Iinterpreted identically to "`child(...)`".

41 younger VALUE

`41 01 <expression> 02`

As in: `younger(...)`

Interpreted identically to "`sibling(...)`".

42 elder VALUE

`42 01 <expression> 02`

As in: `elder(...)`

Returns the object number of the object more recently added to the parent of the object resulting from `<expression>`.

43 prop# INTERNAL VALUE

`43 <property>`

Where `<property>` is a single byte giving the property number.

44 attr# INTERNAL VALUE

```
44 <attribute>
```

Where `<attribute>` is a single byte giving the attribute number.

45 var# INTERNAL VALUE

```
45 <variable>
```

Where `<variable>` is a single byte giving the variable number. 0-239 are global variables, and 240-255 are local to this routine/event/etc.

46 dictentry# INTERNAL VALUE

```
46 <dictionary entry>
```

Where `<dictionary entry>` is two bytes (in low-byte/high-byte order) giving the address of the entry in the dictionary table.

47 text# INTERNAL STATEMENT

```
47 <text address>
```

Where `<text address>` is three bytes (in lowest-to-highest byte order) giving the address of the entry in the text bank.

48 routine# INTERNAL STATEMENT or VALUE

```
48 <routine address>
```

Where `<routine address>` is two bytes giving the indexed address of the specified routine.

49 debugdata# INTERNAL DATA

Is followed by data that is helpful to the engine at runtime—not visible in, for example, the debugger's code window.

E.g., local variable name:

```
49 45 <byte> <data>
```

Where `<byte>` is a single byte giving the number of following `<data>` bytes, which give the name of the next local variable as an ASCII string. Read by the debugger; ignored by the engine.

4A object# INTERNAL VALUE

```
4A <object number>
```

Where `<object number>` is two bytes giving the number of the specified object.

4B value# INTERNAL VALUE

```
4B <number>
```

Where `<number>` is two bytes giving the specified constant value.

4C eol# INTERNAL TOKEN

End-of-line marker.

4D system INTERNAL STATEMENT or VALUE

4D 01 <value> 02 4C[78]

As in: `system(<value>)`

[78] Pre-v2.3 omitted the `eol#` marker ($4C).

Calls the system-level function designated by `<value>`. (See *The Hugo Programming Manual* for further elaboration on the `system` statement.)

Obsolete usage:[79]

```
4D <value>
```

Where <value> is some Hugo data type giving the number of the system function to call.

Source: **herun.c** – `RunSystem()`

4E notheld GRAMMAR TOKEN

4F multinotheld GRAMMAR TOKEN

50 window STATEMENT

```
window n

     50 <value> 4C

window left, top, right, bottom

     50 <v1> 19 <v2> 19 <v3> 19 <v4> 4C

window

     50 4C
```

window 0

```
     50 4B 00 00 4C
```

Where `<value>` or `<vn>`, if present, gives a number of lines or screen coordinate. All instances of the `window` statement are followed by a code block except for "`window 0`". (See

[79] Not implemented post-v2.2.

The Hugo Programming Manual for further elaboration on the `window` statement.)

(Prior to v2.4, the third syntax, i.e., "`window`" alone, complied as "`50 4C`" in v2.3 or simply "`50`" in early versions, followed by a code block, was the only usage. The result was a window beginning at the top of the screen, reaching down to the current cursor row at the termination of the block, and protected then from scrolling of the bottom/main window.)

Source: **herun.c** – `RunWindow()`

51 random VALUE

```
51 01 <expression> 02
```

As in: `random(...)`

Returns a random value between 1 and `<expression>`.

52 word VALUE

```
52 0E <expression> 0F
```

As in: `word[...]`

Returns the dictionary address of `word[<expression>]`.

53 locate STATEMENT

```
53 <value> 4C
53 <value> 19 <value> 4C
```

As in:

```
locate x
locate x, y
```

Where <value> is the column position to reposition the cursor to within the currently defined window. If a second value is given, it represents the new row position.

54 parse$ TOKEN

Read-only engine variable representing the engine parser's internal `parse$` string.

Source: **herun.c** – `RunPrint()`
hemisc.c – `Dict(),GetWord()`

55 children VALUE

```
55 01 <expression> 02
```

As in: `children(...)`

Returns the number of children owned by the object resulting from `<expression>`.

56 in TOKEN

As in: `for <object> in <parent>`

or

```
if <object> [not] in <parent>
```

57 pause STATEMENT

```
57
```

Waits for a keypress. Stores the resulting key value in `word[0]`.

58 runevents STATEMENT

```
58
```

Runs all events in scope.

59 arraydata# VALUE

`array[<expression>]` – element `<expression>` of array `<array>`

```
59 <array> 0E <value> 0F
```

`array[]` – length of array `<array>`

```
59 <array> 0E 0F
```

`array` – address of array `<array>`

```
59 <array>
```

Where `<array>` is two bytes giving the address of the array in the array table.

5A call STATEMENT or VALUE

`5A <value> 4C`[80]

As in: `call <routine address>`

Where `<value>` gives the indexed address of the routine to be called.

5B stringdata# PRINT TOKEN

`5B <`*n*`> <char1> <char2> <char3> ... <char`*n*`>`

Valid only in a print statement. `<`*n*`>` gives the number of characters contained in the print string.

Source: **herun.c** – `RunPrint()`

5C save VALUE

As in: `x = save`

---

80 Pre-v2.3 omitted the `eol#` marker ($4C) when used as a statement.

Calls the engine's save-game procedure (which includes filename input); returns a true value on success, or false on failure.

Source: **`herun.c`** – `RunSave()`

`5D` **`restore`** **`VALUE`**

As in: `x = restore`

Calls the engine's restore-game procedure (which includes filename input); returns a true value on success, or false on failure.

Source: **`herun.c`** – `RunRestore()`

`5E` **`quit`** **`STATEMENT`**

`5E`

Terminates program execution and exits the engine.

`5F` **`input`** **`STATEMENT`**

5F

Prompts for user input, storing the resulting word(s) in the `word[]` array. Unknown (i.e., non-dictionary) words become 0, or ""; the last unknown word is stored in `parse$`.

Source: **`herun.c`** – `RunInput()`

`60` **`serial$`** **`PRINT TOKEN`**

Read-only engine variable representing the compiler-determined serial number.

Source: **`hemisc.c`** – `GetWord()`

61 cls STATEMENT

`61`

Clears the currently defined text window.

62 scripton VALUE

As in: `x = scripton`

Calls the engine's begin-scripting procedure (which includes filename input); returns a true value on success, or false on failure.

Source: **herun.c** – `RunScript()`

63 scriptoff VALUE

As in: `x = scriptoff`

Calls the engine's end-scripting procedure; returns a true value on success, or false on failure.

Source: **herun.c** – `RunScript()`

64 restart VALUE

As in: `x = restart`

Attempts to reload the dynamic game data and restart the game loop; returns a true value on success or false on failure.

65 hex PRINT TOKEN

Signals that the following value should be printed as a hexadecimal number, not as the corresponding dictionary entry.

66 object GRAMMAR TOKEN

(Removed as a token after grammar table is compiled so that "`object`" can refer to the object global variable.)

67 xobject GRAMMAR TOKEN

(Removed as a token after grammar table is compiled so that "`xobject`" can refer to the xobject global variable.)

68 string VALUE

```
68 01 <expr1> 19 <expr2> 19 <expr3> 02
```

As in: `x = string(a, "apple", 8)`

Calls the engine string-writing function to write the dictionary entry `<expr2>` into the array table at the array address given by `<expr1>`, to a maximum of `<expr3>` characters. `<expr1>` is any data type or expression; `<expr2>` is either a value or the `parse$` token ($54); `<expr3>` is optional, and if it is not given, the $02 token comes in place of the second $19.

Source: **herun.c** – `RunString()`

69 array VALUE

```
69 <value>
```

Forces `<value>` to be used as an address in the array table, so that "`array <value>`" can be used as `arraydata#`.

Source: **heexpr.c** – `GetVal()`

6A printchar STATEMENT

```
6A <value1> 19 <value2> 19 ... 4C
```

As in: `printchar 'A', 'B',...`

Outputs a single ASCII character value at the current screen position. Multiple values are separated by $19; the sequence is terminated by $4C.

6B undo VALUE

As in: `x = undo`

Attempts to restore all data changes made since the last typed input; returns a true value on success or false on failure.

Source: **hemisc.c** – `SaveUndo(),Undo()`

6C dict VALUE

`6C 01 <expr1> 19 <expr2> 02`

As in: `x = dict(<array>, <len>)`

Calls the engine dictionary-writing function to write the given string into the dictionary, to a maximum of `<len>` characters. If `<expr1>` is `parse$` ($54), then the value of `parse$` is used; otherwise `<expr1>` is an array address in the array table. If the string is already a dictionary entry, its location is returned. Otherwise, it is appended to the end of the table, and the new location is returned.

Source: **hemisc.c** – `Dict()`

6D recordon VALUE

As in: `x = recordon`

Calls the engine's begin-command-recording procedure (which includes filename input); returns a true value on success, or false on failure.

Source: **hemisc.c** – `RecordCommands()`

6E recordoff VALUE

As in: `x = recordoff`

Calls the engine's end-command-recording procedure; returns a true value on success, or false on failure.

Source: **hemisc.c** – `RecordCommands()`

6F writefile STATEMENT

```
6F <value> 4C
     ...file i/o code block...
```
[81]

As in:

```
writefile <file>
{...}
```

Opens the file named by the dictionary entry `<value>`, erasing it if it previously exists, and runs the following code block. Upon any error, jumps to the end of the file i/o code block and closes `<file>`.

Source: **hemisc.c** – `FileIO()`

70 readfile STATEMENT

```
70 <value> 4C
     ...file i/o code block...
```
[82]

As in:

```
readfile <file>
{...}
```

Opens the file named by the dictionary entry `<value>` and runs the following code block. Upon any error, jumps to the end of the file i/o code block and closes `<file>`.

71 writeval STATEMENT

```
71 <value> 19 <value> 19 ... 4C
```
[83]

[81] Pre-v2.3 omitted the `eol#` marker ($4C).
[82] Pre-v2.3 omitted the `eol#` marker ($4C).

Valid only in a `writefile` block. Writes `<value>` as a 16-bit integer to the currently open file. Multiple values are separated by $19.

72 readval VALUE

As in: `x = readval`

Valid only in a `readfile` block. Reads a 16-bit integer from the currently open file.

73 playback VALUE

As in: `x = playback`

Calls the engines command-playback procedure (including filename input) and attempts to begin command playback from the requested file. If found, player input in `RunGame()` is overridden by commands in the file until end-of-file. Returns true on success, false on failure.

74 colour STATEMENT

Treated identically to $38: `color`.

75 picture STATEMENT

```
75 <value1> 19 <value2> 4C
75 <value1> 4C
```

Attempts to load and display a JPEG-format picture either as resource `<value2>` in resourcefile `<value1>`, or, if `<value2>` is not given, simply as filename `<value1>`. (All `<values>` are dictionary entries.) If there is an error, the `system_status` global variable is set.

76 label# INTERNAL DATA

---

[83] Pre-v2.3 omitted the `eol#` marker ($4C).

77 sound STATEMENT

```
77 [79] <value1> 19 <value2> [19 <value3>] 4C
77 <value1> 4C
```

Attempts to load and play a WAV-format sample as resource `<value2>` in resourcefile `<value1>`. (`<value1>` and `<value2>` are dictionary entries.) If `<value3>` is given, the sample output volume is set to `<value3>` (as a percentage of normal output). If `<value1>` is 0, the current sound is stopped. If there is an error, the `system_status` global variable is set.

78 music STATEMENT

```
78 [79] <value1> 19 <value2> [19 <value3>] 4C
78 <value1> 4C
```

Attempts to load and play a music resource[84] as resource `<value2>` in resourcefile `<value1>`. (`<value1>` and `<value2>` are dictionary entries.) If `<value3>` is given, the music output volume is set to `<value3>` (as a percentage of normal output). If `<value1>` is 0, the current music is stopped. If there is an error, the `system_status` global variable is set.

79 repeat TOKEN

Used by `sound` and `music` statements.

---

[84] Version 2.5 supports MOD, S3M, and XM-format music modules. Version 3.0 and later additionally support MIDI and MP3 files.

INDEX

.HDX file format, 264
.HEX file format, 225
abs (library routine), 202
accented characters, 62, 63, 95, 261
Acquire (library routine), 42, 43, 192
Activate (library routine), 100, 101, 105, 106, 204, 212
adjective (property, compiler-defined), 45, 48, 49, 51, 97, 113, 118, 135, 138, 140, 183
AFTER_PERIOD (library global variable), 180
aliases, 45, 118, 208, 252, 253, 255, 265
already_listed (library attribute), 40, 180
AND_WORD (library constant), 182
AnyVerb (library routine), 92, 192
ARE_WORD (library constant), 182
arguments of routines, 82, 158
Arnold, Julian, 3
array space, 69, 208
arrays, 20, 66, 68, 69, 70, 93, 102, 149, 159, 173, 208, 239, 252, 254, 265, 284
  definition, 68
ASCII characters, 19, 62, 63, 151, 241, 280, 288
assignments, 24
AssignPronoun (library routine), 189, 190, 192
attachable objects, 142
attributes, 20, 39, 40, 41, 42, 45, 48, 49, 50, 51, 53, 117, 121, 122, 132, 149, 195, 198, 199, 208, 216, 242, 251, 252, 255, 264
  aliases, 40
  definition, 39
BANNER (library constant), 20, 93, 181
Baranov, Dmitry, 3
before and after routines, 88, 89, 91, 104, 124, 126, 129, 198, 246
BeOS, 3, 13, 219
BGCOLOR (library global variable), 93, 180
Bijster, Mark, 3
bitwise operators, 65
Blasius, Volker, 3
Blask, Jonathan, 3
BOLD_OFF (font style mask constant), 62, 93, 182
BOLD_ON (font style mask constant), 62, 93, 182
Bostock, Gerald, 3
Bowes, Cam, 3
Brown, Jason, 3
CalculateHolding (library routine), 94, 192, 193
CancelScript (library routine), 103, 205
cant_go (library property), 44, 185
capacity (library property), 42, 44, 46, 97, 132, 183, 192
Cardenas, Daniel, 3
CArt (library routine), 191, 211
Cebrian, Jose Luis, 3
CenterTitle (library routine), 193, 197
character class, 132
character scripts, 102, 103, 181, 205
  routines, 94, 95, 102, 103, 104, 126, 205, 206, 211
CheckReach (library routine), 193
classes
  definition, 48
clothing (library attribute), 39, 179
command-line, 7, 9, 13, 15, 16, 28, 213, 250
comments, 3, 25, 31, 32
  multiple-line, 25
compiler
  directives, 27, 31, 251
  errors, 25, 32
  invocation, 9, 13, 14, 28
  limit settings, 12, 13, 16, 28, 30
  precompiled headers, 6, 11, 28, 216, 217
compiler internal data structures, 252
compiling, 9, 13, 14, 28
component class, 136
compounds, 113
conditional compilation, 28, 209, 217
constants, 19, 20, 22, 23, 43, 52, 54, 55, 56, 59, 62, 65, 66, 68, 93, 94, 150, 155, 160, 162, 170, 175, 176, 182, 187, 195, 208, 251, 254, 255, 273, 275, 280
  enumerating, 55
container (library attribute), 39, 40, 98, 110, 121, 136, 179, 183, 184
Contains (library routine), 105, 193
contains_desc (library property), 45, 97, 98, 185
counter (library global variable), 93, 94, 98, 105, 126, 181, 195, 196, 259
CThe (library routine), 129, 191, 192, 210
cursor_column (display object property), 45, 146, 147, 187

cursor_row (display object property), 45, 146, 187
CustomError (library routine), 121, 181, 190, 194
customerror_flag (library global variable), 181
d_to (library property), 44, 185
daemons (see also fuses), 100
DarkWarning (library routine), 194, 198
data types, 19, 21, 30, 31, 43, 52, 55, 57, 58, 59, 65, 157, 158, 228, 229, 230, 266, 272, 281, 287
Deactivate (library routine), 101, 105, 106, 204, 212
debugger, 2, 3, 5, 11, 213, 263
debugging, 10, 11, 16, 28, 33, 58, 118, 122, 123, 125, 210, 211, 213, 223, 227, 264
DEF_FOREGROUND (color constant), 60, 93, 182
DEF_SL_FOREGROUND (color constant), 60, 93, 182
DEFAULT_FONT (library global variable), 93, 180
DeleteWord (library routine), 194
desc_detail (library property), 45, 186
DESCFORM_F (printing format mask constant), 182
DescribePlace (library routine), 17, 94, 194
dictionary entries, 11, 12, 13, 55, 56, 70, 72, 160, 197, 214, 241, 254, 290, 291
dictionary table, 20, 72, 80, 112, 167, 209, 213, 226, 230, 233, 241, 279
direction class, 131
disambiguation, 122, 189, 210
display object, 34, 45, 146, 147, 151, 187
  properties, 187
door class, 136, 137
door_to (library property), 44, 98, 137, 186
DOS, 3, 7, 9, 11, 61, 63, 79, 213
do-while loops, 75, 77, 161, 176
DOWN_ARROW (library constant), 182
Duchesne, Gilles, 3
Dyer, Jason, 3
e_to (library property), 44, 131, 185
endflag (global variable, compiler-defined), 127, 189
EndGame (junction routine), 54, 121, 128, 227
engine globals (compiler-defined), 180
engine internal data structures, 259
engine properties (compiler-defined), 183
ENTER_KEY (library constant), 182
enterable (library attribute), 39, 98, 179, 184, 229
ESCAPE_KEY (library constant), 182
event table, 247, 251
event_flag (library global variable), 104, 181
events, 247
  global, 95, 247
exclude_from_all (library property), 44, 183
ExcludeFromAll (library routine), 189, 194
expressions, 24, 56, 65, 68, 74, 213, 215, 229, 260, 263, 264, 266
  conditional, 77, 163
female (library attribute), 39, 120, 132, 179
FILE_CHECK (library constant), 150, 170, 183
files
  reading, 149, 150, 170, 171, 177, 183, 229, 289, 290
  writing, 149, 150, 170, 177, 178, 183, 229, 289, 290
FindLight (library routine), 20, 65, 68, 94, 195
FindObject (junction routine), 95, 118, 121, 122, 210, 227
Font (library routine), 62, 93, 182, 195
font style mask constants, 182
for loops, 76
FORMAT (library global variable), 180, 182, 196, 197, 202, 221
found_in (library property), 20, 42, 43, 44, 47, 88, 122, 183, 184
fuses (see also daemons), 101
Future Boy! (Hugo game), 16, 220
game loop, 54, 126, 127, 128, 129, 170, 171, 257, 258, 286
Garza, Miguel, 3
GetInput (library routine), 195
GMD, 3
grammar definition, 5, 107, 108, 113, 235
grammar table, 113, 126, 226, 235, 236, 237, 245, 255, 258, 274, 287
GROUPPLURALS_F (printing format mask constant), 182
hasgraphics (display object property), 45, 146, 154, 187, 260
hasvideo (display object property), 146, 187, 261
Hello, Sailor!, 18, 237, 238
her_obj (library global variable), 181, 193
HERE_WORD (library constant), 182

hexadecimal numbers, 58, 163, 286
hidden (library attribute), 40, 179, 187
higher (library routine), 22, 202
him_obj (library global variable), 181, 193
holding (library property), 42, 44, 97, 132, 158, 167, 178, 184, 192, 193
hours:minutes, 233
HoursMinutes (library routine), 195
Hugo Library, 2, 36, 210
Hugo License, 2
hugofix.g (library file), 6, 28
hugofix.h (library file), 6, 28, 217
hugolib.h (library file), 5, 6, 28, 34, 39, 42, 43, 59, 60, 62, 79, 85, 93, 95, 97, 99, 102, 103, 104, 114, 118, 120, 121, 122, 123, 126, 127, 128, 130, 146, 150, 151, 160, 170, 187, 188, 197, 206, 210, 216, 217
identical objects, 94, 122, 139, 141, 142
IF Archive, 3
if-elseif, 74, 161
ignore_response (library property), 45, 186
in_scope (library property), 44, 101, 122, 184, 200
in_to (library property), 44, 131, 185
IN_WORD (library constant), 182
Indent (library routine), 196
INDENT_SIZE (library global variable), 180, 196
Inform, 2, 4, 220
Init (junction routine), 18, 93, 126, 139, 171, 227
initial_desc (library property), 44, 184, 186, 187, 201
InList (library routine), 196
InsertWord (library routine), 196
inv_desc (library property), 45, 186, 201
IS_WORD (library constant), 182
IsorAre (library routine), 191
IsPossibleXobject (library routine), 196
it_obj (library global variable), 181, 193
ITALIC_OFF (font style mask constant), 182
ITALIC_ON (font style mask constant), 182
Jenness, Jeff, 3
Jones, Doug, 3
junction routines, 117, 127, 128, 189, 226, 227
key_object (library property), 45, 97, 124, 125, 186
Kinder, David, 3
known (library attribute), 39, 122, 179, 199, 212, 213
Lash, Bill, 3
last_object (library global variable), 181
LEFT_ARROW (library constant), 182
legal information, 2
library files, 3, 10, 16, 20, 27, 28, 50, 82, 111, 148, 188, 217
light (library attribute), 20, 39, 60, 66, 94, 130, 135, 179, 181, 194, 195, 199
light_source (library global variable), 181, 195, 199
limit settings (compiler), 12, 13, 16, 28, 30
linelength (display object property), 45, 60, 146, 187
Linux, 3, 7
list_contents (library property), 44, 184
LIST_F (printing format mask constant), 182, 197
list_nest (library global variable), 181
ListObjects (library routine), 181, 197, 202
living (library attribute), 39, 179
location (global variable, compiler-defined), 92
lockable (library attribute), 39, 45, 98, 108, 179, 186
locked (library attribute), 39, 42, 77, 78, 86, 87, 145, 179
long_desc (library property), 42, 44, 48, 49, 97, 130, 131, 135, 184, 194
lower (library routine), 203
MacDonald, Alan, 3
Macintosh, 3, 5, 6, 7, 63, 219
Main (junction routine), 18, 19, 94, 100, 108, 126, 127, 137, 227, 258
MATCH_FOREGROUND (color constant), 60, 182
MatchPlural (library routine), 129, 191
MatchSubject (library routine), 192
mathematical operators, 64
MAX_RANK (library global variable), 180
MAX_SCORE (library global variable), 180
MAX_SCRIPTS (library constant), 182, 205
MAX_WORDS (library constant), 182
MAXALIASES (compiler limit setting), 12, 208
MAXARRAYS (compiler limit setting), 12, 208
MAXATTRIBUTES (compiler limit setting), 12, 208, 253
MAXCONSTANTS (compiler limit setting), 12, 208

MAXDICT (compiler limit setting), 12, 13, 209
MAXDICTEXTEND (compiler limit setting), 13, 72, 73, 160, 209, 226
MAXEVENTS (compiler limit setting), 13, 209
MAXFLAGS (compiler limit setting), 13, 209
MAXGLOBALS (compiler limit setting), 12, 208
MAXLABELS (compiler limit setting), 13, 209
MAXLOCALS (compiler limit setting), 12, 208
MAXOBJECTS (compiler limit setting), 13, 16, 30, 31, 209
MAXPROPERTIES (compiler limit setting), 13, 209
MAXROUTINES (compiler limit setting), 13, 209
Mayo, Cena, 3, 220
McGrew, Jesse, 3
Menichelli, John, 3
Menu (library routine), 181, 197, 206
MENU_BGCOLOR (library constant), 183
MENU_SELECTBGCOLOR (library constant), 183
MENU_SELECTCOLOR (library constant), 183
MENU_TEXTCOLOR (library constant), 183
menuitem (library array), 181
Merrick, Iain, 3
Message (library routine), 197
misc (library property), 45, 184
mobile (library attribute), 39, 144, 179
mod (library routine), 162, 203
mouse input, 151, 219
MOUSE_CLICK (library constant), 151, 182
moved (library attribute), 39, 40, 179
MovePlayer (library routine), 143, 194, 198, 211
multiple lines, 24, 31
music resources, 155, 166
  MIDI, 152, 291
  MOD/S3M/XM, 152, 166, 291
  MP3, 152, 291
n_to (library property), 44, 47, 48, 86, 131, 137, 185
name (property, compiler-defined), 33, 48, 85, 158
ne_to (library property), 44, 185
need_newline (library global variable), 181
needs_repaint (display object property), 147, 187
Nelson, Graham, 2, 4, 220
Newland, Jim, 3
newsgroups
  rec.arts.int-fiction, 3, 220
  rec.games.int-fiction, 3, 220
Nichols, Jerome, 3
NO_AUX_MATH (library compilation flag), 206
NO_FUSES (library compilation flag), 206
NO_MENUS (library compilation flag), 206
NO_OBJLIB (library compilation flag), 206
NO_RECORDING (library compilation flag), 206
NO_SCRIPTS (library compilation flag), 206
NO_STRING_ARRAYS (library compilation flag), 206
NO_VERBS (library compilation flag), 206
NO_XVERBS (library compilation flag), 206
NOINDENT_F (printing format mask constant), 182, 196
NORECURSE_F (printing format mask constant), 182, 197
noun (property, compiler-defined), 20, 24, 42, 43, 45, 46, 51, 97, 108, 113, 118, 138, 183
number_scripts (library global variable), 181
NumberWord (library routine), 99, 198
nw_to (library property), 44, 185
object (global variable, compiler-defined), 34, 43, 53, 89, 109, 110, 117, 167, 168, 180, 245, 287
object library (objlib.h), 130, 131, 136, 137, 138, 139, 141, 142, 145, 213
object specifications (grammar), 109, 110
object table, 12, 37, 242, 244, 251, 255, 261, 262, 265
object tree, 16, 34, 35, 36, 37, 38, 44, 49, 51, 122, 130, 143, 157, 161, 163, 166, 171, 174, 184, 212, 215
ObjectIs (library routine), 199
ObjectisKnown (library routine), 122, 189, 199
ObjectisLight (library routine), 64, 199
objects
  definition, 33, 37, 40
objects (global variable, compiler-defined), 34, 54, 117, 180, 226, 242, 252, 265

objlib.h (library file), 5, 50, 93, 94, 111, 130, 185, 197, 206
ObjWord (library routine), 199
obstacle (library global variable), 181
old_location (library global variable), 94, 181
oldword (library array), 181
ON_WORD (library constant), 182
open (library attribute), 39, 41, 42, 46, 48, 49, 65, 77, 78, 97, 122, 170, 177, 179, 186
openable (library attribute), 39, 49, 98, 179, 186
order of operations, 64
order_response (library property), 45, 123, 128, 186
out_to (library property), 44, 185
override_indent (library global variable), 181
packing list, 4, 9
Palm, 3
Parse (junction routine), 115, 118, 119, 126, 199, 227, 233
parse$, 71, 72, 110, 112, 119, 120, 129, 160, 168, 174, 175, 228, 233, 234, 275, 283, 285, 287, 288
parse_rank (library array), 181
parse_rank (library property), 44, 181, 184, 212
ParseError (junction routine), 119, 120, 128, 227, 234
parser
  engine parser, 118, 283
parser errors, 128, 194, 233, 234
parsing, 13, 14, 111, 115, 119, 121, 189, 190, 196, 211, 212, 233, 234
PauseScript (library routine), 103, 205
Penney, Jason C., 3
Perform (junction routine), 124, 125, 127, 227
picture resources (graphics, images), 153, 154, 169
  JPEG, 152, 290
Pini, Giacomo, 3
platform (library attribute), 39, 40, 121, 136, 179, 183, 184
player_person (library global variable), 133, 180
Plotkin, Andrew, 3
plural (library attribute), 133, 191, 192
plural objects, 39, 139, 140, 141, 145, 179
Pocket PC (WinCE), 3
pointer_x (display object property), 146, 151, 187
pointer_y (display object property), 146, 151, 187
Pontious, Andrew, 3
postfix operators, 66, 67
pow (library routine), 203
prefix operators, 66, 67
PreParse (library routine), 119, 189, 199
preposition (property, compiler-defined), 138
PrintEndGame (library routine), 121, 189, 199
printing format mask constants, 182
printing text, 22, 56, 57, 58, 59, 60, 61, 63, 76, 167, 175, 180, 186, 238, 241, 245, 251, 274, 275, 284
  formatting, 56, 61, 62, 83
  printing numbers, 58, 66, 67, 68, 80, 84, 167
  special characters, 62, 63, 95, 261
printing to an array, 175, 275, 277
PrintScore (library routine), 189, 200
PrintStatusLine (library routine), 94, 126, 200
prompt (global variable, compiler-defined), 54, 93
pronoun (library property), 44, 45, 132, 184
PROP_OFF (font style mask constant), 62, 182
PROP_ON (font style mask constant), 180, 182
properties, 20, 42, 43, 51, 83, 86, 87, 90, 91, 127, 148, 187, 188, 201, 209, 212, 231, 244, 251, 252, 256
  additive, 91
  aliases, 45
  compiler-defined (engine properties), 33, 43, 48, 85, 108, 138, 140, 158
  complex, 43, 244, 245, 252
  definition, 43, 88, 91
  routines, 43, 83, 86, 90, 91, 127, 148, 187, 188, 201, 209, 231, 244, 251, 256
property table, 12, 47, 242, 244, 245, 251, 252
PropertyList (library routine), 200
punctuation (parser), 112
PutInScope (library routine), 184, 200
quiet (library attribute), 40, 179
random numbers, 170, 175, 212, 213, 228, 239, 240, 282
ranking (library array), 181
Ravindran, Vikram, 3
reach (library property), 12, 44, 75, 184, 193

react_after (library property), 44, 127, 185
react_before (library property), 44, 127, 185
readable (library attribute), 39, 179
removals, 112
RemoveFromScope (library routine), 184, 201
replace_pronoun (library array), 181
replacement (of routines, classes, objects), 50, 82
resource.h (library file), 6, 153, 154, 155
resources, 152, 153, 154, 155, 166, 169, 174, 249, 290, 291
ResumeScript (library routine), 103, 205
return values, 18, 20, 54
  default, 86
RIGHT_ARROW (library constant), 182
Roberts, Mike, 2, 220
room class, 50, 130
routines
  addresses, 21, 22, 43, 52, 88, 158, 236, 244, 279, 284
  definition, 82
RunScripts (library routine), 94, 95, 103, 104, 126, 205, 206
s_to (library property), 44, 131, 137, 185
savefile format, 262
scenery class, 135
Schmidl, Gunther, 3
score (library global variable), 19, 93, 180, 181, 200
screenheight (display object property), 45, 146, 147, 154, 187
screenwidth (display object property), 45, 146, 148, 154, 187
Script (library routine), 102, 103, 205, 211
scriptdata (library array), 181
se_to (library property), 44, 185
self (global variable, compiler-defined), 54, 95, 99
serial$, 71, 174, 229, 275, 285
SetObjWord (library routine), 189, 201
setscript (library array), 181
shell game (shell.hug), 6, 37, 51, 60, 93, 94
Sherwin, Robb, 4
short_desc (library property), 20, 42, 44, 45, 97, 184, 185, 186, 187, 201
ShortDescribe (library routine), 201
size (library property), 42, 43, 44, 46, 85, 185
SkipScript (library routine), 104, 205
SL_BGCOLOR (library global variable), 93, 180
SL_TEXTCOLOR (library global variable), 93, 180
sound resources, 155, 174
  wave files, 152, 174
speaking (library global variable), 95, 112, 181
SpeakTo (junction routine), 122, 123, 227
special words, 126, 226, 248
special words table, 248
SpecialDesc (library routine), 201, 202
static (library attribute), 13, 20, 39, 98, 108, 129, 130, 135, 145, 155, 179, 208, 233
statusline_height (display object property), 45, 147, 187
STATUSTYPE (library global variable), 93, 180, 200
string arrays, 70, 71, 72, 110, 174, 204
  routines, 71, 72, 80, 81, 203, 204
StringCompare (library routine), 71, 72, 80, 81, 203, 204
StringCopy (library routine), 71, 72, 203
StringDictCompare (library routine), 72, 204
StringEqual (library routine), 71, 204
StringLength (library routine), 71, 204
StringPrint (library routine), 71, 72, 80, 204
sw_to (library property), 44, 185
switchable (library attribute), 39, 179
switchedon (library attribute), 39, 179
synonyms, 113
system.h (library file), 6, 175
system_status (global variable, compiler-defined), 54, 117, 153, 175, 180, 290, 291
TADS, 2, 220
Tate, Christopher, 4
Tessman, Dean, 4
text
  color, 59, 159
  formatting, 56, 61, 62
  Latin-1 encoding, 62, 63
  special characters, 63, 95, 261
TEXTCOLOR (library global variable), 19, 93, 180
them_obj (library global variable), 181, 193
Tilford, Mark J., 4
title_caption (display object property), 45, 146, 187
transparent (library attribute), 20, 40, 179, 224
Turnbull, Colin, 3

type (library property), 44, 131, 185
u_to (library property), 44, 185
UNDERLINE_OFF (font style mask constant), 62, 182
UNDERLINE_ON (font style mask constant), 62, 182
unfriendly (library attribute), 39, 179
Unix, 3, 5, 7, 9, 11, 13, 14, 16, 61, 63, 79, 213, 261
UP_ARROW (library constant), 182
variables
  global, 53
    compiler-defined (engine globals), 53, 54, 60, 92, 95, 99, 109, 127, 141, 190, 245, 287
    enumerating, 56
  local, 53
Vece, Paolo, 4
vehicle class, 137, 138
verb stub routines, 6, 188, 217
verblib.g (library file), 5, 28, 107, 111
verblib.h (library file), 5, 28, 43, 188, 197
verbosity (library global variable), 181
verbroutine (global variable, compiler-defined), 54, 89, 127, 141, 245
verbroutines, 5, 6, 43, 44, 89, 104, 110, 117, 123, 127, 176, 178, 180, 183, 188, 190, 227
  DoAsk, 133, 134, 188
  DoAskQuestion, 188
  DoBrief, 188
  DoClose, 188
  DoDrink, 188
  DoDrop, 123, 140, 188
  DoEat, 89, 90, 188
  DoEmpty, 188
  DoEnter, 107, 188
  DoExit, 107, 188
  DoGet, 43, 90, 91, 98, 107, 108, 110, 111, 116, 123, 124, 125, 129, 140, 188, 235, 236, 245
  DoGive, 133, 134, 188
  DoGo, 87, 188
  DoHello, 188
  DoHit, 188
  DoInventory, 124, 188
  DoListen, 188
  DoLock, 98, 188
  DoLook, 140, 188
  DoLookAround, 188
  DoLookIn, 188
  DoLookThrough, 188
  DoLookUnder, 97, 188
  DoMove, 188
  DoOpen, 137, 188
  DoPutIn, 90, 140, 188
  DoPutOnGround, 188
  DoQuit, 188
  DoRecordOnOff, 188
  DoRestart, 188
  DoRestore, 188
  DoSave, 108, 188
  DoScore, 188
  DoScriptOnOff, 188
  DoShow, 133, 134, 188
  DoSit, 188
  DoSuperbrief, 188
  DoSwitchOff, 188
  DoSwitchOn, 188
  DoTakeOff, 107, 188
  DoTalk, 188
  DoTell, 133, 134, 188
  DoUndo, 188
  DoUnlock, 188
  DoVague, 107, 108, 143, 188
  DoVerbose, 188
  DoWait, 104, 181, 188
  DoWaitforChar, 188
  DoWaitUntil, 188
  DoWear, 188
verbs, 92
verbstub.g (library file), 6
verbstub.h (library file), 6, 188, 217
VerbWord (library routine), 202
video resources (movies), 155
  AVI, 152
  MPEG, 152
visited (library attribute), 39, 40, 94, 179
w_to (library property), 44, 131, 185
WhatsIn (library routine), 40, 180, 197, 201, 202
when_closed (library property), 45, 186, 201
when_open (library property), 45, 97, 98, 186, 201
while loops, 75, 269, 270
window.h (library file), 6, 148
windowlines (display object property), 45, 146, 187
windows, 45, 61, 146, 147, 148, 154, 159, 165, 187, 281, 286
Windows (Microsoft Windows), 3, 5, 6, 7, 9, 11, 14, 147, 152, 153, 213, 214, 219, 261
word array, 70, 77, 78, 114, 115, 151, 164, 168, 194, 195, 196, 201, 233, 239, 251, 259, 282, 283, 285
workflag (library attribute), 40, 180
worn (library attribute), 39, 179, 199
xobject (global variable, compiler-defined), 190, 287

xverbs, 92

YesorNo (library routine), 202

ABOUT THE AUTHOR

Kent Tessman is a filmmaker and accidental game designer.

THE GENERAL COFFEE COMPANY PRESS
Toronto, Canada