HUGO V2.5

An Interactive Fiction Design System

PROGRAMMING MANUAL

Revised April 1999

TABLE OF CONTENTS

I. INTRODUCTION *4*

I.a. Legal Notes **4**

I.b. (Less Legal Notes) **5**

I.c. Names and Acknowledgments **6**

I.d. Packing List **6**

I.e. Manual Conventions **7**

I.f. Getting Started **8**

I.g. Compiler Switches **9**

I.h. Limit Settings **9**

I.i. Directories **10**

I.j. The Hugo Engine **11**

II. A FIRST LOOK AT HUGO *11*

II.a. Hello, Sailor! **12**

II.b. Data Types **12**

II.c. Multiple Lines **14**

II.d. Comments **16**

II.e. Compiler Errors **16**

II.f. Compiler Directives **17**

Example: Command-Line Compiling 20

III. OBJECTS *20*

III.a. The Object Tree **21**

III.b. Attributes **25**

III.c. Properties **27**

III.d. Classes **32**

IV. HUGO PROGRAMMING *35*

IV.a. Variables **35**

IV.b. Constants **36**

IV.c. Printing Text **38**

IV.d. More Control Characters **42**

Example: Mixing Text Styles 44

IV.e. Operators and Assignments **45**

IV.f. Efficient Operators **47**

IV.g. Arrays and Strings **49**

Example: Managing Strings 53

IV.h. Conditional Expressions And Program Flow 54

V. ROUTINES AND EVENTS 61

V.a. Routines 61

V.b. Property Routines 64
Example: "Borrowing" Property Routines 67

V.c. Before and After Routines 68
Example: Building a Complex Object 71

V.d. Init and Main 72

V.e. Events 73
Example: Building a Clock Event 73

VI. FUSES, DAEMONS, AND SCRIPTS 74

VI.a. Fuses and Daemons 74
Example: A Simple Daemon and a Simpler Fuse 76

VI.b. Scripts 77

VI.c. A Note About the `event_flag` Global 79

VII. GRAMMAR AND PARSING 79

VII.a. Grammar Definition 80

VII.b. The Parser 84

VIII. JUNCTION ROUTINES 87

VIII.a. Parse 88

VIII.b. ParseError 88

VIII.c. EndGame 90

VIII.d. FindObject 91

VIII.e. SpeakTo 92

VIII.f. Perform 93

XI. THE GAME LOOP 94

X. ADVANCED FEATURES 95

X.a. The Display Object 95

X.b. Windows 96

X.c. Reading and Writing Files 97

XI. RESOURCES 99

XI.a. Pictures 100

XI.b. Sound and Music 102

APPENDIX A: SUMMARY OF KEYWORDS AND COMMANDS 102

APPENDIX B: THE LIBRARY (HUGOLIB.H) *124*
ATTRIBUTES **124**
GLOBALS, CONSTANTS, AND ARRAYS **124**
PROPERTIES **127**
ROUTINES **131**
APPENDIX C: LIMIT SETTINGS *146*
APPENDIX D: PRECOMPILED HEADERS *148*
APPENDIX E: THE HUGO DEBUGGER *149*

I. INTRODUCTION

Hugo is a system for designing, programming, and running sophisticated interactive fiction, or text adventures. It is the result of an attempt to further extend the concepts developed in earlier, similar systems in order to make interactive fiction programming less cryptic and more accessible to designers. Hugo owes much to the original Infocom format (particularly with regard to its internal data tables) as well as to Graham Nelson's publicly distributed Inform compiler (and its syntactic interpretation of the Infocom format and straightforward grammar definition).

The best advice to be given for learning Hugo is probably to print or otherwise have handy the source listing of `SAMPLE.HUG`, and to refer to it throughout; examples of almost all of Hugo's features may be found in the source of the sample game.

Author e-mail (The General Coffee Company Film Productions):
<generalcoffee@geocities.com>

Hugo Home Page:
http://www.geocities.com/hollywood/academy/5976/hugo.html

(As of this revision)

I.a. Legal Notes

Programs created using the Hugo Compiler are the property of the individual author. Note, however, that the library files are copyright by Kent Tessman, the creator of Hugo, as is the Hugo Engine.

The use of the Hugo library files and the distribution of the Hugo Engine are authorized so long as all transactions are non-commercial and free of charge (except in cases where any charge is to cover the cost of distribution), and that the library files and engine are not distributed in a modified form.

For those interested in the commercial distribution of a program created with the Hugo Compiler, please contact Kent Tessman for permission.

NOTE: Since the Hugo Compiler and Engine are provided free of charge, there is no warranty for their use.

I.b. (Less Legal Notes)

This supplementary, less-official section is meant to clarify my intentions as to legal usage of Hugo, and what it means for users who may want to distribute their games. First off, let me say that, yes, I do want to be able to maintain some discretion over what is done with my work, and the above phrasing is intended to reserve me that ability. Hugo is more than "just" a compiler--it's a complete design and runtime environment, so distribution may involve more than just a simple `.HEX` file (which, even though it might have been built with the Hugo Library written by me, isn't really a cause for concern on my part).

Here's a quick informal overview of how I see the various types of distribution:

1. *Freeware.* I don't really have any concerns, even if the Hugo Engine is being distributed as part of a free package. (Although it might be nice to know about this, just so far as wanting to help ensure that proper instructions, updated information, etc. were included.) As far as using the Hugo Library goes: I wrote it for this express purpose, so that people would use it in making their own games. Freeware distribution is certainly something I fully encourage. The source to the Hugo Library, of course, cannot be distributed in modified form unless it is expressly indicated that it was a.) written by Kent Tessman, and b.) subsequently modified and distributed by someone else.

2. *Shareware.* Again, I don't think this really concerns me, either. (Although, again, I would like to be aware of shareware distribution. My only real objection would be to something that is entirely morally reprehensible--I probably don't need to go into details. In that case, I'd probably tell you to write your own damned library and interpreter.) The shareware market for IF is sadly depressed, but writing a good game takes a lot of talent and hard work--and if Hugo authors wanted to try to generate some shareware revenues, I would wish them luck.

3. *Commercial software.* This is about the only sticking point I can think of, and it's not much of one. It's unfortunately pretty unlikely that someone could market a wildly successful piece of commercial Interactive Fiction. On the other hand, I do believe that Hugo is capable of creating some pretty eye- (and ear-) catching games. And on the other other hand, especially with commercial software, what you'd be distributing would likely be more than half written by other people (i.e., me, the library contributors, and any porter(s), since I'd be impressed if someone wrote a game that, in its source-lines count, rivaled the 35,000+ lines of code in the engine and the library). But even in a case like this, I would expect any individual license to give the author the freedom to sell *n* copies of the game without involving me or anyone else in any (however minor) participation.

I.c. Names and Acknowledgments

Those who have taken upon themselves the (sometimes trying, I'm sure) task of porting Hugo to various platforms--aside from the author's own 16-bit DOS, 32-bit DOS, and Windows ports--are:

David Kinder	Amiga
Colin Turnbull	Acorn Archimedes
Bill Lash	Linux (plus Solaris OS, etc.)
Gerald Bostock	OS/2

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 by way of criticisms both generous and deservedly direct.

More than a few words of appreciation must be given to Volker Blasius who (now with help from David Kinder) has had the substantial responsibility of maintaining the Interactive Fiction Archive at *ftp://ftp.gmd.de*--one of the key resources for Hugo programmers and a primary hub of material for contributors to (and readers of) the newsgroups *rec.arts.int-fiction* and *rec.games.int-fiction*.

Thanks also to those whose comments and suggestions have contributed to making Hugo as useful and usable as it is: Dr. Jeff Jenness, Vikram Ravindran, Jesse McGrew, Paolo Vece, Daniel Cardenas, Cam Bowes, Mark Bijster, Jose Luis Cebrian, John Menichelli, Jerome Nichols, Jason Dyer, and Jason Brown. Acknowledgment and thanks are also due Graham Nelson, whose Inform language helped give shape to Hugo's early syntax and structure.

Special thanks to Julian Arnold and Jim Newland, members of the ad hoc Hugo What-If? Committee. They've both made numerous valuable contributions to Hugo both in terms of suggestions for the language itself and for user library improvements and extensions--to be more specific would surely be to overlook something invaluable.

Finally, my brother Dean Tessman has been a well-used resource with his willingness to test-drive multi-100K e-mail attachments of executables and to engage in ongoing back-and-forth discussions on programming minutiae.

I.d. Packing List

A number of files are part of the basic Hugo package:

(NOTE: Throughout this manual, the default naming convention is for MS-DOS/Windows. As Hugo becomes available for other systems, file naming conventions may vary, and any machine-specific documentation should document those variations.)

HC.EXE	Hugo Compiler
HE.EXE	Hugo Engine
HD.EXE	Hugo Debugger
HDHELP.HLP	Debugger help file
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)
SAMPLE.HUG	Sample game source code
SHELL.HUG	Source code to build on

And two sets of files that, depending on user-specified settings, are optionally included by HUGOLIB.H, VERBLIB.H and VERBLIB.G:

HUGOFIX.H	Debugging routines
HUGOFIX.G	Debugging grammar
VERBSTUB.H	Additional verb routines
VERBSTUB.G	Additional verb grammar

An additional Hugo source file demonstrates the ability to create precompiled headers:

HUGOLIB.HUG	To create a linkable version of HUGOLIB.H

The latest release of Hugo is available through anonymous FTP from *ftp.gmd.de* in if-archive/programming/hugo. Distribution of any of the Hugo files is authorized only with permission of the author as per *Legal Notes*, above.

The .HUG, .H, and .G files are text files and must be downloaded as such; the executables are binary files.

(FORMATTING NOTE: The above files are properly formatted for a standard tab stop of 8 spaces; if the formatting appears incorrect, adjust the tab size on your editor.)

I.e. Manual Conventions

The following conventions are used in this manual:

<parameter>	for required parameters

`[parameter]`	for optional parameters
`FILE`	for specific filenames
`FunctionName`	functions, etc.
`'token'`	tokens, keywords
`...`	for omissions

I.f. Getting Started

Type

```
hc
```

without any parameters to get a full listing of available compiler options and specifications.

The MS-DOS syntax for running the compiler is

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

It is not 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. With the compiler executable, library files, and sample game source code all in the current directory, 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.

I.g. Compiler Switches

A number of switches may be selected via the invocation line. The available options are:

-a	Abort compilation on any error
-d	compile as an .HDX debuggable executable
-e	Expanded error format
-f	Full object summaries
-h	compile in .HLB precompiled Header format
-i	display debugging Information
-l	print Listing to disk as <sourcefile>.LST
-o	display Object tree
-p	send output to standard Printer
-s	print compilation Statistics
-t	Text to listfile for spellchecking
-u	show memory Usage for objectfile
-v	Verbose compilation
-x	ignore switches in source code

Most Hugo programming will probably make us of the -l switch in order to record compile-time errors.

The -z switch may, on some configurations, increase compilation speed by inhibiting normal messaging (i.e., “Compiling...lines of...” and “...percent complete”).

I.h. Limit Settings

Also included on the invocation line, after any switches and before the sourcefile, may be one or more limit settings. These settings are for memory management, and limit the number of certain types of program elements, such as objects and dictionary entries.

To list the settings, type:

```
hc $list
```

To change a non-static limit, type:

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

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* for a complete listing of valid limit settings.

I.i. 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 are:

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

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.

I.j. The Hugo Engine

Having compiled the sample game, run it by invoking

```
he sample
```

at the command line. Again, it is not necessary to specify the extension. The engine assumes `.HEX` if none is given.

(NOTE: 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.)*

II. A FIRST LOOK AT HUGO

There are a couple of basic concepts to become oriented to in order to begin working with Hugo.

First of all, most programming in Hugo will involve the creation of what are called "objects". Quite literally, these represent the "objects" or elements of the game universe: people, places, and things.

The bulk of the rest of a Hugo program is 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 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 returning some kind of value as a result.

(The idea of return values is an important one and, while sometimes puzzling to novices, is actually quite uncomplicated. Often a particular function will be referred to as "returning true" or "returning false"--all this means is that it returns either a non-zero value (usually 1) or a zero value, almost always to indicate success or failure. A program will constantly be checking the return values of a variety of routines and commands to determine if a particular operation was successful in order to decide what to do next. Of course, a return value can be any integer value; a routine that adds together two supplied values, a and b, may return the sum a+b.)

For those familiar with the common programming languages C and BASIC, Hugo strongly resembles a hybrid of the two. Individual objects and routines--as well as conditional blocks--are enclosed in braces as in C, but unlike C (and like BASIC), a

semicolon is not required at the end of each line, and the language itself is considerably less cryptic. Keywords, variables, routine and object names, and other tokens are not case-sensitive.

The goal in designing Hugo was to make programming as intuitive to facilitate both initial development and subsequent debugging.

II.a. Hello, Sailor!

The grand tradition of programming texts has an introduction to a new programming language detailing how to print the optimistic phrase "Hello, world" as an example of the particular language's form and substance.

In the equally grand tradition of interactive fiction, we'll start with the rallying cry "Hello, Sailor!". Don't worry too much about the syntax below; this is meant mainly as a familiarization with what Hugo looks like.

```
routine main
{
     print "Hello, Sailor!"
     return
}
```

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 example since there isn't much required in the way of initialization.)

The `Main` routine is automatically called by the engine. It 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 an order to return from the routine (i.e., exit it) so that we don't strand the program waiting for an input, which is the normal order of Hugo business.

II.b. Data Types

All data in Hugo is represented in terms of 16-bit integers, treated as signed (-32768 to 32767) or unsigned (0 to 65535) as appropriate. 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 `suitcase, 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)

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"` *is stored.)*

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--see above)

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. Dictionary entries are addresses in the dictionary table, with the null 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. 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 engine functions, so that

```
FindLight(room)
```

and

```
parent(object)
```

are also valid integer data types.

Routine addresses are also stored as 16-bit integers. However, those versed 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. Such is not the case, since the routine address is actually an indexed representation of the absolute address.

NOTE: The 16-bit format of a routine address (or the address of a property routine, to be discussed below), can obtained via the address operator '&', as in:

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

(where `x` is a variable).

II.c. 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)
```

The space at the end of the first line is necessary because the compiler automatically trims leading spaces from the second line.

String constants, such as in the above print statement, are an exception in that they do not require the '\' character at the end of each line.

```
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" error when it overruns the object/routine/event boundary looking for a resolution to the odd number of quotation marks.

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

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

and

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

translate into

```
     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--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."}
```

(See sections below on code formatting to see exactly what these constructions represent.)

II.d. Comments

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 \!
```

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.

II.e. Compiler Errors

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

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

and halts compiler execution.

A non-fatal error typically looks like:

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

Also, the compiler may issue warnings in the form:

```
<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, since the compiler often paraphrases and rebuilds the source code into a more rigid format before building the line.

II.f. Compiler Directives

A number of special commands may be used to determine a.) how the source code is read by the compiler, or b.) what special output will be generated at compile time.

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 -ils
```

the first line in every new program, which will automatically print out debugging information, a statistical summary, and any 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.

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 command.

(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 D* on *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 develop add-ons to 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.

To set and clear flags, use

```
#set <flagname>
```

and

```
#clear <flagname>
```

respectively.

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 <flag>`" and "`#if undefined <flag>`" to test if objects, properties, routines, etc. have previously been defined.

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>`".

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.

Example: Command-Line Compiling

On the author's machine, running under MS-DOS, the compiler executable `HC.EXE` is in a directory called `C:\HUGO`. The library files are in `C:\HUGO\LIB`, and the source code for the game Spur is in `C:\HUGO\SPUR`.

It's possible to call the compiler to compile Spur with a number of different options, including setting compiler flags to include the HugoFix debugging library and verb stub routines (i.e., what could otherwise be accomplished with "`#set DEBUG`" and "`#set VERBSTUBS`" in the source), and printing all debugging information, the object tree, and statistics to a file. (Assume that the current directory is `C:\HUGO` and that none of the switches or compiler flags are set in the source.)

```
hc -iols #debug #verbstubs @source=spur @lib=lib spur
```

This makes use of all the possible command-line option types, including multiple switches, flag settings, and directory specifications.

III. OBJECTS

Objects are the building blocks of any Hugo program. Anything that must be accessible to a player during the game--including items, rooms, other characters, and even directions--must be defined as an object.

The basic object definition looks like this:

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

As an example, a suitcase object might be defined as:

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

The enclosing braces are needed even if the object definition has no body. The only data attached to the suitcase object are--from right to left--a name, an identifier, and membership in the basic object class.

The compiler assigns the object labeled `<objectname>` the next sequential object number. That is, if the first-defined object is the "nothing" object (object 0), then the next-defined object, whatever it is, is given the object number 1; the one after that is 2, etc. This is academic, however, as a programmer need never 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 "name" (or name property) is provided, the compiler automatically gives it the name "`(<objectname>)`", i.e., `<objectname>` in parentheses.

(The compiler automatically creates an object called "display" as the last defined object. The display object can be used to get information about the engine's output display. See the section on the display object below under *"Advanced Features"*.)

III.a. The Object Tree

In order for objects to have a position 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 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. This is the root of the object tree, upon which all other objects are based.

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.

(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.)

Here is an example of an object tree:

```
Nothing
|
Room
|
Table------Chair------Book------Player
|                      |
Bowl                   Bookmark
|
Spoon
```

A number of 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 a particular `<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
|          |         |
...        Player    ...
```

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 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.)

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

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

will return true if <object> is in <parent> (or false if 'not' is used).

To initially place an object in the object tree, use

```
in <parent>
```

in the object definition, or, alternatively

```
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, 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' specifier.

```
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 suitcase "suitcase"
{
     in emptyroom
}
```

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

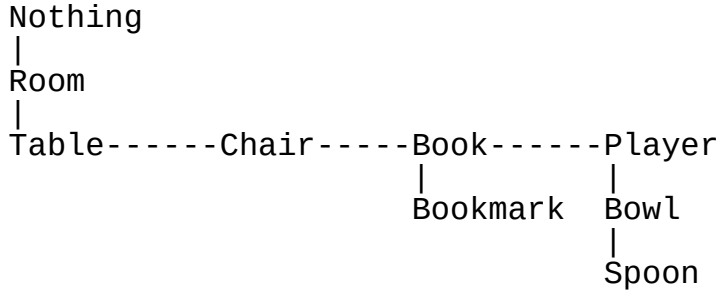
```
move <object> to <new parent>
```

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:



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.

III.b. Attributes

Attributes are essentially qualities that every object either does or doesn't have. They are most useful for qualifying or disqualifying objects for or from consideration in any given instance.

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
`unfriendly`	if a character is unfriendly
`openable`	if an object can be opened
`open`	if it is open
`lockable`	if an object can be locked
`locked`	if it is locked
`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 *(NOTE: container and platform are mutually exclusive)*
`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)
`workflag`	for system use
`special`	for miscellaneous use

Some of these attributes are actually the same attribute with different names. This 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), so:

```
attribute visited alias moved
```

In this case, an object which is visited is also, by default, moved. It is expected that attributes which are aliased will never both need to be checked under the same circumstances.

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.

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.

Attributes can also be read using the '`is`' and '`is not`' structures. As a function,

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

returns true (1) if `<object>` is (or is not, if '`not`' is specified) `<attribute>`. Otherwise, it returns false (0).

To give the suitcase object the appropriate attributes, expand the object definition to include

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

Now, the following equations hold true:

```
suitcase is openable = 1
suitcase is open = 0
suitcase is locked = 0
```

III.c. 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.

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 addresses of executable code. The maximum number of attached values is undefined, but manageability and efficiency suggest eight or less.

These are some valid properties as they would appear in an object definition (using property names defined in `HUGOLIB.H`):

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

The nouns property contains 4 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 single values representing the address of the attached routine.

The `before` property is a special case. This complex property 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 `globals 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`. (There is also an `after` routine, which is checked after the verb routine has been called.)

(Note for clarity: the `Art` routine from `HUGOLIB.H` prints the appropriate article, if any, followed by the name of the object. 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.)

All of this may be a little confusing for now. There will be more on property routines later. For now, think of a property as simply containing a value (or set of values).

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.

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
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)
type	to identify the type of 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
size	for holding/inventory
capacity	“ “ “
holding	“ “ “
reach	for limiting object accessibility
list_contents	for overriding normal listing
door_to	for handling “Enter <object>“
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
parse_rank	for differentiating like-named objects
exclude_from_all	for interpreting “all” in inputs
misc	for miscellaneous use

(For a detailed description of how each property is used, see *Appendix B: The 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

Note that while some of these, namely `screenwidth` through `title_caption`, are defined as constants in the library, they are still usable as property references, since both property numbers and constants are simple integers.)

Property names may again be aliased by

```
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
```

A 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 suitcase "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 engine 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".

(NOTE: 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
```

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. (Running the debugger with runtime warnings enabled will help spot instances like this.)

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
```

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
```

III.d. Classes

Classes are essentially objects that are specifically intended to be used as prototypes for one or more similar objects. 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 printing command--the braces are unnecessary.)

The definition of an object in a 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.

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.)

Since a class is basically an object, it is possible to define an object using a previous object as a class even though the previous object was not explicitly defined as a class. Therefore,

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

is perfectly valid.

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 first from `<class1>`, then from `<class2>`, and so on (or even `<object1>`, `<object2>`, etc.).

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 the `OBJLIB.H` file, 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 any uses of the class for other objects.)

IV. HUGO PROGRAMMING

IV.a. Variables

Hugo supports two kinds of variables: global and local. Either type simply holds a 16-bit integer, 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:

```
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>[ = <starting 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>[ = <starting 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 starting 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 a verb 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
`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` and `xobject` 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). 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 `verbroutine` variable holds the address of the routine specified in the grammar table and corresponding to the entered command; 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.

The `objects` variable can be set by the player, 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
```

outputs:

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 definite 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
}
```

gives: `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 “+x”, “-x”, “*x”, or “/x”, where x is a constant integer value. To count by twos:

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

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

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

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

Otherwise the specifier "`constants`" is implied as the default.

IV.c. Printing Text

Text can be printed 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 control character ("\") is handled specially. It modifies how the character following it in a string is treated.

`\"`	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:

```
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.)

NOTE: These control-character combinations are valid for printing only; they are not treated as literals, as in, for example, expressions involving dictionary entries.

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.

NOTE: 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 control-character combinations are still usable with these print statements, but since each command 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 usable 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.

The standard color set with corresponding values and constant labels is:

COLOR	CONSTANT 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

(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. 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.

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 trim them as necessary.

IV.d. More Control Characters

As listed above, the following are valid control characters 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 control characters 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 MS-DOS (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

(Print style can also be changed using the `Font` routine in `HUGOLIB.H`. Font-change constants can be combined as in:

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

where the valid constants are `BOLD_ON`, `BOLD_OFF`, `ITALICS_ON`, `ITALICS_OFF`, `UNDERLINE_ON`, `UNDERLINE_OFF`, `PROP_ON`, and `PROP_OFF`.)

Special characters can also be printed via control characters. 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	(¥)
\-	em dash	(—)

\#xxx any ASCII character where xxx represents the three-digit ASCII number of the character to be printed
e.g. "\#065" will print an 'A' (ASCII 65)

Example: Mixing Text Styles

```
! Sample routine to print various typefaces and colors:

#include "hugolib.h"

routine PrintingSample
{
     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 MS-DOS ports, for example, uses color changes instead of actual typeface changes, and does not support proportional printing.

IV.e. Operators and Assignments

Hugo allows use of all standard math operators:

+	addition
-	subtraction
*	multiplication
/	integer division

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

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

evaluate respectively to 3 and 5 if x is 1, and 2 and 4 if x 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
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

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)
```

(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 or a property.

```
<variable> = <expression>

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

In certain cases, the compiler may allow a statement where the left-hand side of the assignment is non-modifiable. I.e.

```
Function() = <expression>
```

or

```
<object>.#<property> = <expression>
```

may be compiled, but such statements will force a run-time error from the Hugo Engine.

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 these usages, 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:

```
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>` 16-bit words, so that, for example,

```
array test_array[10]
```

initializes ten 16-bit words 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 would return a meaningless value. (Trying to assign it by mistake would likely overwrite something important, like the next-defined array.)

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)*x, "Hello!", FindLight(location)
```

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. However, single elements of arrays are valid arguments.

Significantly, it is 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)
```

to produce the output "socks", even though `v` is an argument (i.e., local variable), and not an array. The line "`print array v[2]`" tells the engine to treat `v` as an array address, not as a discrete value.

Array strings are also possible, and Hugo provides a way to store a dictionary entry in an array as a series of characters using the '`string`' command:

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

(The `<max. length>` provision is required because the engine has no way of checking for array boundaries.)

For example,

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

will store up to 10 characters from `word[1]` into `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 or null character after the string itself.

For example,

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

will store up to 10 characters of `word[1]` in the array `a`, and return the length of the stored string to the variable `x`.

(The engine variables '`parse$`' and '`serial$`' may be used in place of the dictionary entry address; see the section below on "*Junction Routines: ParseError*" for a description.)

The 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>])
```

For example, if 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--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 a complement to the ‘`string`’ command, the ‘`dict`’ 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. In this case, the new words will have to exist in the dictionary, and must be written using '`dict`'. However, a guideline for programmers is that there should be a limit to how many new words the 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.)

Example: Managing Strings

```
#include "hugolib.h"

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 must be treated 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.

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...}
```

NOTE: 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,

```
if <expression1>
     for (...<expression2>...)
          if <expression3>
               ...conditional code block...
```

will not be.

(Technically speaking, the compiler will misunderstand the end of the '`for`' loop construction because the enclosing conditional code block expects to end with the '`for`' expression. In turn the '`for`' expression does not properly differentiate the end of the conditional loop. The result would likely be a stack overflow error in the engine because the engine will continually nest the execution of recursive '`for`' loops until it runs out of stack space.)

The proper way to structure that same section of code would be:

```
if <expression1>
{
     for (...<expression2>...)
     {
          if <expression3>
               ...conditional code block...
     }
}
```

NOTE: The best advice is to rely on braces to clarify code structure whenever using such complex constructions. This applies particularly to mixing '`if`', '`for`', '`while`' and "`do-while`" expressions, especially when recursive function calls are involved. While the results may appear as intended, the method to produce them is incorrect, and any long-running such construction is almost guaranteed to crash the stack.

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...
```

In this case, the engine quickly performs an evaluation that 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`’. The same rules for bracing multiple-line code blocks apply as with ‘`if`’ (as well as for every other type of conditional block).

NOTE: Cases do not “fall through” to the following case. Think of cases following the first as being ‘`elseif`’ statements rather than ‘`if`’ statements; once a true case has been found, subsequent cases are ignored.

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

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.

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 "; number i
```

First, the engine executes the assignment setting "`i = 1`". Then, it executes the `print` statement. 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`.

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 expression. It may be a routine that modifies a global variable, for example, which is then tested by the '`for`' loop.

(A second form of the ‘`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 suitcase
     print i.name
```

will print the names of each object in the `suitcase` object.)

The easiest way to picture the first form of a Hugo ‘`for`’ loop is that

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

translates to the equivalent of

```
<assignment>
[while] <expression>
{
     ...conditional code block...
     <modifier>
}
```

which in turn translates the equivalent of

```
<assignment>
:<label1>
[if] <expression>
{
     ...conditional code block...
     <modifier>
     jump <label1>
}
```

(On the other hand, that isn’t a particularly easy way to picture anything, and, in its awkwardness, perhaps justifies the existence of non-threatening ‘`while`’, “`do-while`”, and ‘`for`’ loops.)

The benefit in knowing how a Hugo loop breaks down into a slip knot of ‘`if`’ and ‘`jump`’ statements is that it is easier to monitor program flow using the Hugo Debugger (see *Appendix E*).

As is now obvious by the above (possibly confusing) illustration, Hugo 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. (Care should also be taken with using 'jump'--it is generally far preferable to use alternatives, as there exists a potential for overflowing the engine's stack when not using standard looping constructions.)

It is also important to recognize--particularly with 'select' and 'while' or "do-while" statements--that the expression is tested each time the loop executes, or, in the case of a 'select' statement, for each corresponding case. The significance of this is seen in the following example

```
select test.prop_routine
     case 1
          {...}
     case 2
          {...}
     case 3
          {...}
```

where prop_routine returns a value from 1 to 3. The property routine will be executed 3 separate times, once for each 'case' statement. If prop_routine has some other effect, such as modifying a global variable or printing output, then this will also occur 3 times.

If such an effect would be undesirable, try

```
local test_val                  ! set up a local variable
test_val = test.prop_routine    ! and assign it
select test_val
     case 1
          {...}
     ...
```

so that test.prop_routine is called only once.

A similar case would be where

```
select random(3)
     case 1:  {...}
     case 2:  {...}
     case 3:  {...}
```

would result in something akin to:

```
if random(3) = 1:      {...}
if random(3) = 2:  {...}
if random(3) = 3:  {...}
```

In other words, a different random value would be evaluated each time. A better choice would be:

```
local b
b = random(3)
select b
     case 1:  {...}
     ...
```

One final keyword 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). ‘break’ 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
if box not in livingroom, garage
if a ~= 1, 2, 3
```

are translated into

```
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.

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). (A number of routines have occurred in previous examples, but here is the formal explication.)

A routine is defined as

```
routine <routinename> [(<argument1>, <argument2>, ...)]
{
     ...
}
```

once again ensuring the 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> [(<argument1>, <argument2>, ...)]
```

For example,

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

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.

`TestRoutine` can be called several ways:

```
TestRoutine(suitcase)
```

will (assuming the `suitcase` 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(suitcase)
```

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(suitcase, 5)
```

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 `Double`.

An interesting side-effect of a null (0) return value can be seen using the ‘`print`’ command. 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 null string, or ""), the '`print`' command is actually displaying

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

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

V.b. Property Routines

Property routines are slightly more complex than those 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

```
size 10
```

an object may contain the property

```
size
{
     return x + 5
}
```

Trying to read `object.size` in either case will return an integer value.

Here's another example. Normally, if `<object>` is the current room, then `<object>.n_to` would contain the object number of the room to the north. 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."

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

(For those wondering why the true 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.)

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> =
{
     ...
}
```

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 re-assignment of property values to routines and vice-versa. A property routine is given a "length" of one 16-bit word, 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 16-bit word of data--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.

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.

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 <default value>
```

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.

If `<object>.before` 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 a clearer example using the `suitcase` object we have been developing:

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

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

Each of these examples will return true, thereby overriding the engine's default operation (see the section on *"The Game Loop"*). 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
          {"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 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 specify a verbroutine.

For example,

```
before
{
     xobject
     {
          ...
     }
}
```

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

If this 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. This can be particularly undesirable in the case of location `before`/`after` properties, where a non-specific response will be triggered even for '`save`', '`restore`', etc.

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(!). But `AnyVerb(location)` will only equal the `location` global if the verbroutine is not an xverb.)

Example: Building a 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
}
```

And for a challenge: how could the cabinet be converted into, say, a secret passage into another room?

ANSWER: Add a `door_to` property, such as:

```
door_to secondroom        ! a new room object
```

The cabinet can now be entered via: "go cabinet", "get into cabinet", "enter cabinet", etc.

V.d. Init and Main

At least two routines are typically part of every Hugo problem: "`Init`" and "`Main`". (The latter is required. The compiler will issue an error if no `Main` routine exists.)

`Init`, 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 begin the game.

`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.

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.

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
{
     ...
}
```

for global events, and

```
event [in] <object>
{
     ...
}
```

for events attached to a particular object. (The '`in`' is optional, but may be useful 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).

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.

Example: Building a Clock Event

Suppose that there is a clock object in a room. Here is a possible routine:

```
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";
          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 house--i.e., at any point in 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.)

VI. FUSES, DAEMONS, AND SCRIPTS

While all of the above mentioned elements of Hugo are programmed into the internal code of the engine, the means of running fuses, daemons, and scripts are written entirely in Hugo itself and contained in the library (HUGOLIB.H).

VI.a. 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. It 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>)
```

or

```
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--which decrements the timer--and executes the following conditional block if s`elf.timer` is 0.

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) and stop it with `Activate(tickingbomb, 25)` and `Deactivate(tickingbomb)`.

VI.b. 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. It is up the programmer not to overflow this limit.

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(<obj>, <number>)] = &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 the `<number>` 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)

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.)

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.c. A Note About the `event_flag` Global

The library routines--particularly the `DoWait...` verb routines--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.

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 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                                 DoUnLock
     * lockable "with" held                     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.

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

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.

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 fairly 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 matched, this 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 lower case.

The next step is to process the three 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 basically 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 the Parse routine in `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")
```

A maximum of thirty-two words is allowed. The period is in each case converted to the null dictionary entry (““, 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.

VIII. JUNCTION ROUTINES

Because, simply put, the engine is unaware of such things as attributes, properties, and objects in anything but a technical sense, 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 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
`location`	location of the player
`verbroutine`	the verb routine address
`endflag`	if not false (0), call `EndGame`
`prompt`	for 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
`adjective`	adjective(s) for referring to object
`article`	“a”, “an”, “the”, “some”, etc.

(As well as 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 these will likely not be satisfactory for most programmers. Therefore, `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.a. 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 words (by the engine).
2. The `Parse` routine, if it exists, is called.
3. Control returns to the engine for grammar matching.

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; 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.

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.b. 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 a special variable called ‘`parse$`’, usable only in a `print` statement (or in conjunction with ‘`string`’), 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.c. 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
(3	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.d. 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. 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.)

VIII.e. 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>)
```

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.

```
order_response
{
     if verbroutine = &DoGet
          "I would, but my back is too sore."
     else
          return false
}
```

VIII.f. 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>)
```

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.)

For example, the 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)

>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, and object.before, then finally verbroutine if none of those returns true.)

XI. THE GAME LOOP

This the paradigm 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.

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.

(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.

Setting the global endflag at any point to a non-zero value will terminate the game loop and run the `EndGame` junction routine.

NOTE: Undo information recalled by 'undo' is saved each turn only during the `Main` routine (including any commands or functions called within, such as events, fuses and daemons, or character scripts) and verb routines (unless the verb was an xverb). It is therefore recommended that no other routines change any significant game data, because it will not be recoverable with 'undo'.

X. ADVANCED FEATURES

X.a. The Display Object

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

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.

X.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 top-left corner (`l`, `t`) and the bottom-right corner (`r`, `b`), where these coordinates are character coordinates on the full-screen
`window` `{...}`	Redraws the last-defined window

Each of these usages 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`".

An windowing library file exists called `WINDOW.H` which defines a `window_class` and the associate 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.

X.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 the '`save`' and '`restore`' functions return a true value to `x` if successful, or a false value if for some reason they fail. In either case, the entire set of game data--including object locations, variable values, arrays, attributes, etc.--is saved or restored, respectively.

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. RESOURCES

The engine allows a Hugo program to access external data (called resources) compiled into a specially formatted file called a resourcefile. 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 v2.5 supports JPEG graphic files, RIFF/WAV audio samples, and MOD/S3M/XM music modules as resources.

For example, here is an imaginary example resourcefile compiled on a Windows 95/NT platform:

```
resource "gameres1"
{
     "c:\hugo\graphics\logo.jpg"
     "h:\data\scenic panorama.jpg"
     "h:\data\background.jpg"
     "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 95/NT, 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 `system_status`, which may be tested after a resource operation. If `system_status` is non-zero (signifying normal status), it will contain one of the following values:

-1	STAT_UNAVAILABLE
101	STAT_NOFILE
102	STAT_NORESOURCE
103	STAT_LOADERROR

XI.a. Pictures

A picture is displayed as a resource in a resourcefile using:

```
picture "<resourcefile>", "<picture>"
```

For example,

```
picture "gameres1", "logo"
```

(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, ensures that one or more lines at the top of the screen 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. Because of the potential inaccuracy of this method, it is generally recommended to always specify the resourcefile name.)

XI.b. Sound and Music

Sounds and music are played using the Hugo statements:

```
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).

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>[5]
```

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 (or 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>` (from `HUGOLIB.H`) are:

0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Dark gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Light yellow
15	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 song (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 ASCII value of the key is stored in `word[0]`.

PICTURE

DESCRIPTION: Load and display a picture in the current text 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.

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. See the section above on *"Reading and Writing Files"*.

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.

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 and calls the `Init` routine.

SYNTAX: `x = restart`

NOTE: '`Restart`' 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 by calling the engine's restore routine.

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 by calling the engine's save routine.

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 by calling the engine's transcription routine.

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 and play an audio sample (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>)`

Function	Label	Description
11	READ_KEY	Read keypress (key value)
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

(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 LIBRARY (HUGOLIB.H)

ATTRIBUTES

`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
`unfriendly`	if a character is unfriendly
`openable`	if an object can be opened
`open`	if it is open
`lockable`	if an object can be locked
`locked`	if it is locked
`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 *(NOTE:* `container` *and* `platform` *are mutually exclusive)*
`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, for example, a `WhatsIn` listing)
`workflag`	for system use
`special`	for miscellaneous use

GLOBALS, CONSTANTS, AND ARRAYS

GLOBALS:

The first 10 globals are pre-defined by the compiler:

oobject	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 input line
objects	the total number of objects
player_person	first (1), second (2), or third (3)
MAX_SCORE	total possible score
MAX_RANK	up to x levels of player ranking
FORMAT	specifies text-printing format
DEFAULT_FONT	initially 0; could be, for example, PROP_ON
STATUSTYPE	0=none, 1=score/turns, 2=time
TEXTCOLOR	normal text color
BGCOLOR	normal background color
BOLDCOLOR	color for boldface printing
SL_TEXTCOLOR	statusline text color
SL_BGCOLOR	statusline background color
INDENT_SIZE	for paragraph indenting
AFTER_PERIOD	string of spaces following a full-stop
counter	elapsed turns (or time, as desired)
score	accumulated score
verbosity	for room descriptions
list_nest	used by ListObjects
light_source	in location
event_flag	set when something happens
speaking	if the player is talking to a char.
old_location	whenever location changes
last_object	set by Perform to value of object
obstacle	if something is stopping the player
best_parse_rank	for differentiating like-named objects
customerror_flag	true once CustomError is called
need_newline	true when newline should be printed
override_indent	true if no indent should be printed
number_scripts	number of active character scripts
it_obj	to reference objects via pronouns
them_obj	
him_obj	

her_obj	
general	for general use

ARRAYS:

replace_pronoun[4]	for it_obj, him_obj, etc.
oldword[MAX_WORDS]	for "again" command
scriptdata[48]	for object scripts
array setscript[1024]	the actual scripts
array ranking[10]	in tandem with scoring

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 masks (for setting 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 masks (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	

AND_WORD	("and")	IN_WORD	("in")
ARE_WORD	("are")	IS_WORD	("is")
HERE_WORD	("here")	ON_WORD	("on")

FILE_CHECK for verifying `writefile`/`readfile` operations

(*The following are used only by specific routines:*

ARRAYS:

_temp_array[256]	used by string manipulation functions
menuitem[11]	required by the `Menu` function

GLOBALS:

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., used generally for room objects in order to give a grammatically correct

	description if necessary; also for containers and platforms
`pronoun`	"he", "him", "his" or equivalent, so that an object is properly referred to
`short_desc`	routine; basic "X is here" description
`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`
`long_desc`	routine; detailed description
`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 not be included in the object definition, since `found_in` values are unrelated to "object in parent" relationships
`type`	to identify the type of object, used primarily by class definitions in `OBJLIB.H`
`size`	for holding/inventory purposes, contains a value representing the size of an individual object
`capacity`	contains a value representing the capacity of a `container` or `platform`
`holding`	contains a value representing the current encumbrance of a `container` or `platform`
`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; if access is limited to the object in question only, reach must still contain at least one non-false value (i.e., the parent object itself)
`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
`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 or cleared using `PutInScope` or `RemoveFromScope`

`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-existant) value--used when `FindObject(<obj>, 0)` is called

`exclude_from_all` — returns true if the object should be excluded from actions such as "get all"

`misc` — miscellaneous use

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 room object in direction X, then `x_to` holds the new room object

cant_go — routine; message instead of default "You can't go that way."

For non-room objects only:

`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)

`key_object` — if `lockable`, contains the object number of the key

`when_open`, `when_closed` — routines; short descriptions for `openable` objects

	If they exist, the appropriate `when_open` or `when_closed` routine is called instead of `short_desc` (if an `initial_desc` does not exist, or if the object has been moved)
`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."
`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
`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
`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 (`;`)
`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 (`;`)

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:

`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

Read/writable:

`title_caption`	dictionary entry giving the full proper name of the program (optional)

Defined in `HUGOLIB.H`:

`statusline_height`	of the last-printed status line

(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.)

ROUTINES

VERB ROUTINES:

`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. 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, DoDrop, DoEat, DoEnter,
DoExit, DoGet, DoGive, DoGo, DoHit, DoInventory,
DoListen, DoLock, DoLook, DoLookAround, DoLookIn,
DoLookThrough, DoLookUnder, DoMove, DoOpen, DoPutIn,
DoShow, DoSwitchOff, DoSwitchOn, DoTakeOff, DoTalk,
DoTell, DoUnlock, DoVague, DoWait, DoWaitforChar,
DoWaitUntil, DoWear
```

Here are the non-action verb routines:

```
DoBrief, DoQuit, DoRestart, DoRestore, DoSave, DoScore,
DoScriptOnOff, DoSuperbrief, DoVerbose
```

(NOTE: A set of verb stub routines is also available, including the actions:

```
DoBurn, DoClimb, DoCut, DoDig, DoFollow, DoHelp, 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.:

Routines may be treated as procedures or functions, given the idea that procedures are more like commands, while functions are expected to return a value, as in:

```
Procedure(a, b)
x = Function(y)
if Function()...
```

Library routines that do not return a value are generally meant to be treated as procedures; those that do return a value may be treated as either functions or procedures.

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, 3 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()`

Returning true forces the engine to re-parse the modified input line.

Also calls: `PreParse`, `AssignPronoun` and `SetObjWord`

`ParseError` — called by the engine via:
`ParseError(errornumber, object)`

Returning false signals the engine to print the default error message. Return 2 to force the existing line to be reparsed as is.

May also call: `CustomError`

`SpeakTo` — called by the engine via:
`SpeakTo(character)`

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:

`The` — calling form: `The(object)`

Prints the definite article form of the object name, e.g. “the apple”

`Art` — calling form: `Art(object)`

Prints the 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”

`CArt` calling form: `CArt(object)`

Prints the capitalized indefinite article form of the object name, e.g. “An 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 the formal parameter is specified as true).

`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 null string). Therefore, either of the following usages are valid:

```
CThe(apple)
print " is here."
```

or

```
print CThe(apple); " is here."
```

Other 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 `object` to the specified parent, and returns true. If the object cannot be moved, `Acquire` returns false.

Also calls: `CalculateHolding`

`AnyVerb`

calling form:
`AnyVerb(object)`

Returns object 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 held 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 given 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.

`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, otherwise returns false.

`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 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 death of the player, replace the default with a new `DarkWarning` routine.

`DeleteWord`

calling form:
`DeleteWord(wordnumber[, number])`

Deletes the number of words given by the second argument--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 the player's 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 null 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 the global `counter` represents the time in minutes from 12:00 a.m. If the optional `military` value is given as a true value, time is in 24-hour "military" format.

`Indent`

calling form:
`Indent`

If the `NOINDENT_F` bit is not set in the `FORMAT` mask, `Indent` prints two spaces without printing a newline

`InList`

calling form:
`InList(object, property, value)`

If <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.

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 choices (up to 10) are contained in the menuitem array, with menuitem[0] is the title of the menu. A starting selection number is 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.

(NOTE: 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 specified (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 this instance, MovePlayer moves in that direction from the player object's present location.

If <none> is true, before/after routines are not run.

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.

Returns the object number of the player object's new parent.

NOTE: 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 null 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 routine defined in the library is empty.

`PrintEndGame`

calling form:
`PrintEndGame(end_type)`

Depending on whether `end_type` is 1 or 2, prints "*** YOU'VE WON THE GAME! ***" or "*** YOU ARE DEAD ***".

`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(obj, property)

Lists the objects held in obj.property (if any), returning the number of objects listed.

PutInScope

calling form:
PutInScope(object, actor)

Makes <object> accessible to <actor>, regardless of their respective locations, and providing that the in_scope property of <object> has at least one empty slot--i.e., one that equals 0. Returns true if successful.

RemoveFromScope

calling form:
RemoveFromScope(object, actor)

Removes <object> from the scope of <actor>. Returns true if successful, or false if <object> was never in scope of <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 `<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.

`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 the absolute value of `<a>`.

`higher`

calling form:
`higher(a, b)`

Returns the higher number of `<a>` or `<b>`.

`lower`

calling form:
`lower(a, b)`

Returns the lower number of `<a>` or `<b>`.

`mod`

calling form:
`mod(a, b)`

Returns the remainder of `<a>` divided by `<b>`.

`pow`

calling form:
`pow(a, b)`

Returns `<a>` to the power of `<b>`. (The return value is unpredictable 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:

(See the earlier section on fuses and daemons for more information.)

`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:

(See the earlier section on character scripts for more information.)

`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 object `<character>`. Returns true if successful.

`ResumeScript` calling form:
`ResumeScript(character)`

Resumes execution of a paused script. Returns true if successful.

`SkipScript` calling form:
`SkipScript(character)`

Skips execution of the script for `<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 objects to use in scripts. 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 which 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`	`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

NOTE: 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 setting using:

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

either in the 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
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 in an entire program
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 stand-alone routines (not property routines) that may be defined

APPENDIX D: PRECOMPILED HEADERS

It is possible to compile 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>"
```

The advantage of doing this is primarily one of faster compilation speed; files that are used over and over again without alteration (such as HUGOLIB.H) may be precompiled so that they are not recompiled every time.

The #link directive must come after any grammar, but before any 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. Compile it via

```
hc -h hugolib.hug
```

in order to generate HUGOLIB.HLB.

Next, change occurrences of

```
#include "hugolib.h"
```

in Hugo programs 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.

NOTE: 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 .HLB headers compiled in separate passes via subsequent #links 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 E: 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.

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 nearly exact approximation of the original source code.

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 that `.HDX` files can be run by the engine, but `.HEX` files cannot be run by the debugger because of the additional data required.)

The MS-DOS convention for running the debugger is:

```
hd <filename>
```

The debugger will begin on the debugging screen. Switch back-and-forth from the actual game screen by pressing Tab.

At 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 HD.EXE--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

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 in color scheme (if applicable), printer, etc.