Roger Firth’s

?

INFORM 6: FREQUENTLY ASKED QUESTIONS

PDF version of the HTML document found at Roger Firth’s IF pages

Introduction

You're reading a set of answers to Frequently Asked Questions about the Inform programming language, intended especially to help those who are novices in this arena. Unless you are planning to write a piece of Interactive Fiction -- a text adventure game -- using Inform, you might as well go right to the *Ifaq* for more general assistance.

Note that the questions here are all about Inform 6. There is almost nothing in these pages about Inform 7 -- Graham's radical new approach to IF authorship.

This FAQ aims to address topics which commonly cause confusion among newcomers. However, you shouldn't set your expectations too high; it isn't an Inform tutorial, it pre-supposes that you've got a copy of the *Inform Designer's Manual* and it assumes at least a little knowledge of computer programming. Very rarely will these answers give you the full story. Generally, they're designed as introductory tasters, providing just enough information to illustrate the general principles, and to point you towards the appropriate material in the definitive *Designer's Manual*.

There are other good sources: make sure that you know about the *Inform Beginner's Guide* (available, like the *Designer's Manual*, from the Inform website), and about these more general FAQ pages:

- The *Ifaq* gives a very brief overview: Interactive Fiction (IF) from ten thousand feet;
- The ifwiki FAQ covers much the same ground, but in greater depth;
- The FAQ for the rec.arts.int-fiction (RAIF) Usenet newsgroup covers the authorship of IF in many systems, not just Inform;

Also, these excellent Q&A sites may resolve your more detailed or sophisticated Inform concerns:

- Jim Fisher's Inform Guide;
- Marnie Parker's Inform Primer and Programming Tips;
- Andrew Plotkin's Handy Inform Tricks;
- David Fisher's summary of Past RAIF topics;
- John Escobedo's record of Inform How Do I ...? postings on RAIF.

Finally, there's also a very old Inform FAQ which relates to Inform 5 and DM2. None of the downloads work and some of the information is unreliable, life having moved on since 1995; tread carefully here.

Many people helped in the creation of this document, sometimes unwittingly; my thanks for all of the assistance. The Inform FAQ is copyright of and maintained by Roger Firth, who is always pleased to receive your comments and contributions. First posted in July 2001 and last updated: 24 May 2006. This PDF version was concocted by Sonja Kesserich, 28 June 2006.

Contents

1 · Setting the Scene

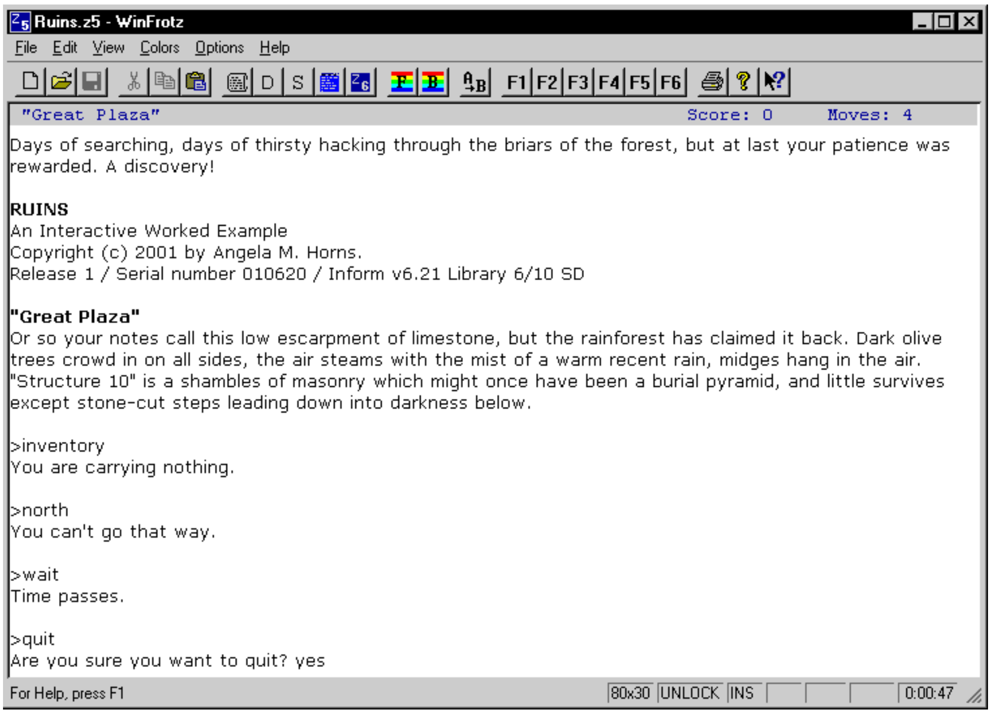
So, what *is* Inform?

From the Introduction to the *Inform Designer's Manual*: "Inform is a system for creating adventure games. It translates an author's textual description into a simulated world which can be explored by readers using almost any computer, with the aid of an interpreter program."

In its simplest possible form, the "author's textual description" looks rather like this:

```
Constant Story "RUINS";
Constant Headline "^An Interactive Worked Example^
                   Copyright (c) 2001 by Angela M. Horns.^";

Include "Parser";
Include "VerbLib";

Object  Forest "~Great Plaza~"
  with  description
       "Or so your notes call this low escarpment of limestone,
        but the rainforest has claimed it back. Dark olive
        trees crowd in on all sides, the air steams with the
        mist of a warm recent rain, midges hang in the air.
        ~Structure 10~ is a shambles of masonry which might
        once have been a burial pyramid, and little survives
        except stone-cut steps leading down into darkness below.",
  has   light;

[ Initialise;
  location = Forest;
  "^^^Days of searching, days of thirsty hacking through the briars of
  the forest, but at last your patience was rewarded. A discovery!^";
];

Include "Grammar";
```

and the "explored ... with an interpreter program" part might be along these lines:



```
 "Great Plaza"                                         Score: 0      Moves: 4
Days of searching, days of thirsty hacking through the briars of the forest, but at last your patience was
rewarded. A discovery!

RUINS
An Interactive Worked Example
Copyright (c) 2001 by Angela M. Horns.
Release 1 / Serial number 010620 / Inform v6.21 Library 6/10 SD

"Great Plaza"
Or so your notes call this low escarpment of limestone, but the rainforest has claimed it back. Dark olive
trees crowd in on all sides, the air steams with the mist of a warm recent rain, midges hang in the air.
"Structure 10" is a shambles of masonry which might once have been a burial pyramid, and little survives
except stone-cut steps leading down into darkness below.

>inventory
You are carrying nothing.

>north
You can't go that way.

>wait
Time passes.

>quit
Are you sure you want to quit? yes
```

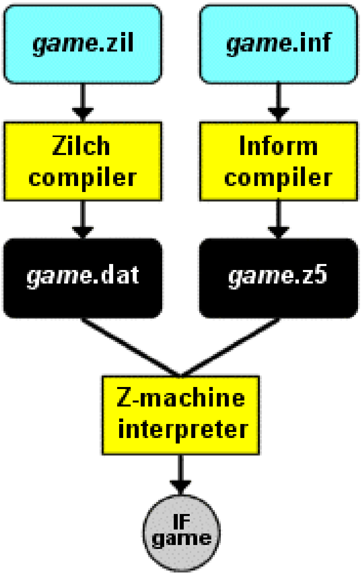
Needless to say, real adventure games are much more exciting -- and much more complex -- than our tiny example. Nevertheless, almost all games look more or less like this, and behave more or less in this manner.

Where do I begin?

Inform is the creation of Graham Nelson (who by day teaches mathematics at Oxford University), and you can't do better than start at his web site. In particular, read the answers to some background questions that you may have, get details of the files that you'll need to download, and find links to other Inform web sites.

For updates on what's new in Inform, visit Roger Firth's Informary

How is Inform related to Infocom?



Infocom was the company, formed in 1979 by ex-MIT students to capitalize on the popularity of Adventure and its imitators, which over the following ten years created more than thirty text adventure games; many of those are highly regarded, and still widely played today. Infocom's games were written in a specially-devised Zork Implementation Language (ZIL) and compiled by Zilch into Z-code. A Z-code game could be played using a Z-machine interpreter program, and many interpreters were written to run on the wide range of hobbyist microcomputers then in vogue.

Eventually, text adventures fell from public favour, Infocom disappeared into Activision, and the specifications of ZIL and the Z-machine were lost. All that remained in general circulation were the Z-code games themselves. In an astonishing feat of reverse-engineering, a group of enthusiasts known as the Infocom Task Force managed in the early 1990s to deduce the architecture of the Z-machine by inspecting the contents of these binary (non-text) files, and they documented their researches in the Z-machine Standards Document.

That specification made it possible to create new Z-machine interpreters, and thus to play the original games on computers which hadn't existed when Infocom was around. There was, however, no way to create new games for the Z-machine until Graham devised Inform. Although the Inform language is, at least superficially, nothing like ZIL, and the Inform compiler is quite different from Zilch, nevertheless the outcome of compiling a source game is the same in both cases -- a file of Z-code which can be played on any Z-machine interpreter. Many Inform programmers view this, the commonality of Z-code between their games and the original Infocom masterpieces, as one of the coolest features of the system.

When did Inform appear?

The first version of Inform appeared in 1993, and the system has been growing steadily in capability and usage ever since.

Version	Date	Compiler	Library
Inform 1	Apr 1993		
Inform 2	??? 1993		
Inform 3	Nov 1993		
Inform 4	Jan 1994		
Inform 5	Jun 1994 ... Jun 1995		
Inform 6	Apr 1996 May 1996 May 1996 Sep 1996 Sep 1996 Dec 1996 Jan 1997 Mar 1997 Apr 1997 May 1997 Aug 1997 Sep 1997 Mar 1998 Dec 1998 Apr 1999 Nov 1999 Feb 2004	6.01 6.02 6.03 6.04 6.05 6.10 6.11 6.12 6.13 - - 6.14 6.15 6.20 6.21 - 6.30	6/1 - - 6/2 - 6/3 6/4 - - 6/5 6/6 6/7 - 6/8 6/9 6/10 6/11
Inform 7	Apr 2006	6.31	6/10N

Looking at this (slightly simplified) chart, you can see how Inform initially evolved quite rapidly, running through five major versions in its first three years. Some of those early versions were fairly primitive; not until Version 6 did it settle into a form closely resembling the system that we use today.

In fact, although the core system didn't change at all between 1999 and 2004, Inform enthusiasts continue to find ways of extending and enhancing that core using a wide variety of techniques. Once you've mastered the basics of Inform, be sure to look at the many **contributed extensions**; also, if you should come across a possible bug in the compiler or the library, here's a list of **well-known problems**.

For a fascinating account of its evolution, see the final section of the Inform Technical Manual

Should you need them, the Archive holds the non-current versions of the Inform 6 Library and some versions of the Compiler. If you have source for any of the missing Compilers (6.01, 6.02, 6.04, 6.05, 6.11, 6.12), *please* upload it to the Archive

Version 7 marks another major leap in Inform's evolution. Rather than being a serial development of what has gone before, Inform 7 offers instead a radical new approach to IF authorship. Stories are created using a simplified form of English; this is automatically processed behind the scenes into a Version 6 game, which is then compiled and run. The Version 6 code is not intended to be read by the author, and depends on a heavily-modified version of the 6/10 library files.

What's the best way to learn Inform?

You have a choice! If you're an experienced programmer, and especially if you're already familiar with other IF design systems, you can follow the seven-stage training plan:

Skim the *Inform Designer's Manual*.
Try some very simple examples of your own.
Read the *Inform Designer's Manual*.
Look at one or two tutorials.
Study the *Inform Designer's Manual*.
Write something slightly more complex.
Consult the *Inform Designer's Manual*.

For when you need just to look up a syntax detail, Roger Firth's Inform in four minutes and InfoLib at your fingertips quick reference summaries may be helpful

What this tells you is: the Fourth Edition of the *Inform Designer's Manual* -- colloquially known as the DM4 -- is far and away the best source of reliable detailed information about programming in Inform. It's well-written, well-formatted, well-indexed; it's, well, a splendid document whose pages you can't know too thoroughly. Even the "Natural Language" chapter -- aimed primarily at translating Inform into languages other than English -- is worth your while reading.

On the other hand, if this whole programming scene is new to you, then you'll probably be better off starting with the *Inform Beginner's Guide*. This takes a much more leisurely stroll through Inform, dwelling lovingly on the basic principles, while completely omitting much of the advanced stuff. Written as a tutorial, it introduces Inform through the design of three simple games; it's mean to be read straight through, not used as a reference manual -- for that, you'll still want to turn to the DM4.

When you first read the DM4, especially if you don't have much programming experience, it can all seem rather daunting. In your early days, the most important sections to know are:

§1.1 to §1.12 (but just browse quickly to get a feel for the language);
§2.1 "Directives", §2.2 "Constants" and §2.3 "Global variables";
§3.1 "Objects", §3.2 "The object tree" and §3.3 "Setting up the tree";
§4 "'Ruins' begun";
§8 "Places and scenery" and
§9 "Directions and the map".

Note that the first chapter of the DM4 gives a complete and detailed definition of the Inform **Language**, while the following four chapters focus on the Inform **Library**. The language itself is not very different from other programming languages; what makes Inform special is the Library, which transforms your general-purpose computer into a dedicated adventure machine. So, to start with, don't worry about understanding all 70 pages in the first chapter; skip past things which seem difficult, and come back to them later. As soon as you can, move on to Chapter II -- Introduction to Designing -- which is where the Library comes into play and adventure games begin to happen.

You can download both the DM4 (2.9Mb PDF file) and the IBG (1.4Mb PDF file) for free from the Inform website.

Does anybody teach Inform?

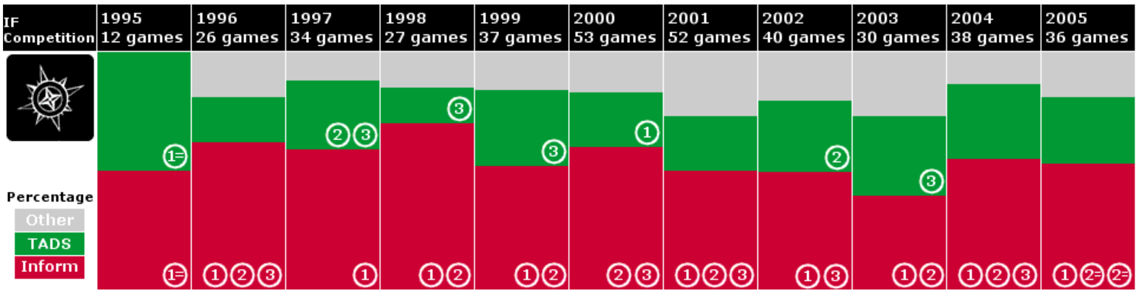
A few US academics have run courses. **Nick Montfort** at the University of Pennsylvania has sample syllabi for a semester-long Workshop in Interactive Fiction course and for a similar one-month intensive course with some different selections. **Dennis Jerz** runs IF classes at Seton Hill University, also in Pennsylvania (as he did in his previous position at the University of Wisconsin-Eau Claire). **Ken Forbus** at the Northwestern University in Illinois taught Computer Game Design in Spring 2003, **Stephen Ramsay** had a Fall 2003 class on Digital Narratives at the University of Georgia, and **Bruce Maxim** at the University of Michigan–Dearborn had a Fall 2004 class on Computer Game Design and Implementation with some Inform content. Not a vast range, maybe, but it certainly adds up to a little academic respectability.

Further afield, Inform figured on a Summer 2004 class on Computer Game Design in New Zealand, and is listed in a public course schedule for the good folks of Saugus, Massachusetts.

How popular is Inform?

It's extremely difficult to know how many people actually use the system. In very round figures, about 800 Inform games have been published, by perhaps 450 different authors. Making some allowance for those who have finished but not published a game, who have one or more unfinished games in progress, or who have dabbled in Inform without producing anything significant... and it's still hard to believe that many more than a thousand people have ever seriously tried to compile an Inform file. We're not exactly a large community.

Having said that, Inform is the most popular of today's industrial-strength IF languages, at least using a few simplistic measures. There have been around 24,000 rec.arts.int-fiction postings with "Inform" in the subject line, as against some 10,000 mentioning "TADS". The Archive holds more than twice as many Inform games as those created using TADS, and the Inform games seem to be downloaded more frequently. In the annual IF Competition, Inform has always accounted for the largest batch of entries, with TADS coming second, and the lesser systems battling over third place.



Although Inform games tend to fare a little better than TADS in the competition, this doesn't mean that TADS is an inferior system -- technically the new TADS 3 is almost certainly better than Inform in many ways -- but rather that Inform has in recent years proven more popular with authors. Some of the reasons for this are probably:

the documentation: the *Inform Designer's Manual* is a remarkable work, and the more modest *Inform Beginner's Guide* complements it nicely;
accessibility: Inform code is apparently more readable and less intimidating on first acquaintance;
the coolness factor: writing games that run on the same interpreters as Infocom's;
portability: the Z-machine is supported in more environments, including -- crucially -- on PalmOS handhelds;
the addons: well over 100 Library extensions contributed to the Archive by the Inform community;
techie features such as inbuilt assembly language support and the simplicity of replacing Library functions;
success breeds success. Initially, Inform attracted a large user base because it was freely available, because it was being very actively developed, and because of the quality of games like "Curses". TADS was at that time shareware, and in somewhat of a state of development limbo; neither of these is still true, but Inform's momentum -- having become the more 'visible' system -- has helped to keep it in the lead.

Which isn't to say that Inform couldn't be improved. There are justifiable complaints about:

the syntax: initially confusing, both to non-programmers *and* those experienced in other programming languages;
the Library: insufficiently modular, difficult to extend, contains too many special cases;
the Z-machine: less cool than frost-bitten, the ageing Z-machine's inadequacies are only partially overcome by Glulx;
timidity: Graham's hesitancy about making changes which, while clearly desirable, would be incompatible with existing code.

However, don't let such niggles put you off: you'll find -- along with those hundreds of other users -- that Inform's good points far outweight any downside.

Where are all these games you mention?

In the 'games' section of the Archive. You'll find source games in ...games/source/inform, compiled (ready to play) games in ...games/zcode and ...games/glulx.

Carl Muckenhoupt's invaluable Baf's Guide to the IF Archive currently lists (and rates) hundreds Inform games.

Michael Baum's excellent Z-code catalog, also in the Archive at ...info, stops in mid-1999. Here's a -- certainly imperfect -- list of about 500 Inform games published between then and end-2003 (thanks to XYZZYnews for the raw data).

Can I play Inform games on my handheld?

Yes! Handheld computing and text adventures seem to be especially well-suited, and there are Z-code interpreters for many small machines.

I've seen Inform games played on the web... can I do that?

Sure: all that you need is a special interpreter program, written in the Java language, which runs in a web browser. You don't have to do anything special to the game, other than upload it -- along with a copy of the Java applet -- to your web site. Here's a starter kit, comprising the applet, a template web page, and a copy of Roger Firth's demonstration game "Cloak of Darkness".

I'm blind -- is there any way I can play Inform games?

I think so. A helpful guy named Warren Scott Dillman has on ongoing IF text-to-speech project, which includes an adapted version of David Kinder's Windows Frotz 2002. If you come across other aids to blind Inform gamers, please let me know.

2 · Preparing to program

What do the various file extensions like 'Z5' signify?

In an operating environment like the PC where filename extensions are widely used, you're likely to come across some of these values:

More information in the DM: §41 §45

Extension	Used for
`.INF`	Files containing Inform source programs.
`.H`	'Header' files containing Inform source statements which are intended to be **#Include**d into `.INF` files.

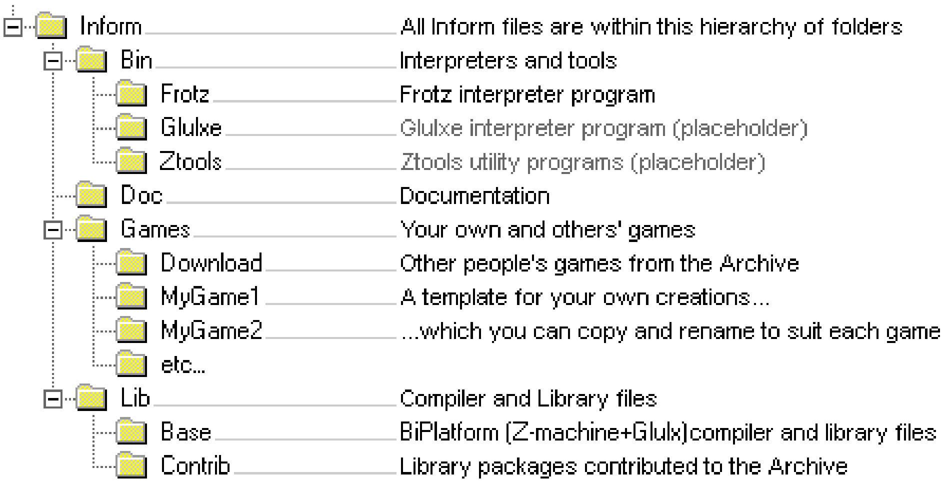
`.Z5`
`.Z8`
	The vast majority of today's compiled Z-machine games: `.Z5` (Infocom's 'advanced' design) is normal unless the game size exceeds 256Kb, whereupon `.Z8` permits an increase to 512Kb.

`.Z3`
`.Z6`
	Infrequent circumstances: `.Z3` (Infocom's 'standard' design) is nowadays created only to suit some restricted-memory interpreters which cannot handle `.Z5` files; `.Z6` (Infocom's 'graphical' design) lacks the development tools and interpreter support for widespread acceptance.

`.DAT`	Infocom games. Note that interpreters use information from *within* the file, rather than the file's extension, to determine its version; any Z-machine interpreter should be able to play a `.DAT` file.
`.ULX`	Inform games which have been compiled to run on the Glulx virtual machine.
`.BLB`	Blorb files: a packaged collection of sounds and/or images which can be invoked from a `.Z6` or `.ULX` game.
`.SAV`	Preserving a game's state, created by the SAVE verb.
`.SCR`	Keeping a copy of everything displayed during a game, created by the SCRIPT ON verb.
`.REC`	Keeping a copy of every command typed while debugging a game, created by the RECORDING ON verb and re-used by the REPLAY verb.

Where should I store the various Inform files?

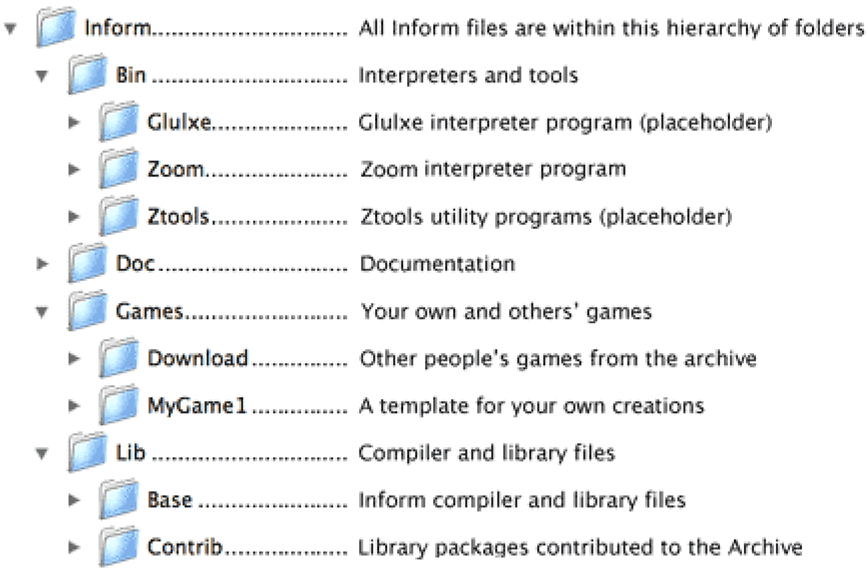
When you start programming in Inform, you need to download various program and data files, and store them in some convenient arrangement of folders or directories. How you do that is entirely up to you; some people simply place everything in a single folder, but we think that too easily becomes confusing. We like to keep the 'system' files -- the Library, compiler and interpreter -- separate from the game files that we're editing, so our preference is for a logical organization of folders. On a PC it looks like this:



```
Inform                All Inform files are within this hierarchy of folders
  Bin                 Interpreters and tools
    Frotz             Frotz interpreter program
    Glulxe            Glulxe interpreter program (placeholder)
    Ztools            Ztools utility programs (placeholder)
  Doc                 Documentation
  Games               Your own and others' games
    Download          Other people's games from the Archive
    MyGame1           A template for your own creations...
    MyGame2           ...which you can copy and rename to suit each game
    etc...
  Lib                 Compiler and Library files
    Base              BiPlatform (Z-machine+Glulx)compiler and library files
    Contrib           Library packages contributed to the Archive
```

If you like this approach, you can download the PC folders; this is a 1Mb compressed file which also contains enough Inform files to get started with.

On a Mac running OS X it looks very similar:



```
▼ Inform............................. All Inform files are within this hierarchy of folders
  ▼ Bin ............................. Interpreters and tools
    ▶ Glulxe......................... Glulxe interpreter program (placeholder)
    ▶ Zoom........................... Zoom interpreter program
    ▶ Ztools......................... Ztools utility programs (placeholder)
  ▶ Doc ............................. Documentation
  ▼ Games............................ Your own and others' games
    ▶ Download....................... Other people's games from the archive
    ▶ MyGame1 ....................... A template for your own creations
  ▼ Lib ............................. Compiler and library files
    ▶ Base .......................... Inform compiler and library files
    ▶ Contrib........................ Library packages contributed to the Archive
```

Here you can download the Mac folders as a compressed file with equivalent contents. Put this file in a temporary location on your Mac and use a tool like StuffIt Expander to unpack it. You'll now have a new Inform folder, that you should place in a suitable location in your hard disk (for example, your home directory).

Or, of course, you can just follow the instructions on Graham's page and fetch the individual files yourself. If you do this, remember to rename the downloaded Library files to have an extension of "`.h`" (for example, rename `Grammar` to `Grammar.h`).

On UNIX, you need to be careful because filenames are case-sensitive. I suggest that you stick to lower case filenames, and then add appropriate symbolic links. For example, one of the Library files is called `verblib.h`; by adding links from `Verblib.h` and `VerbLib.h` to that file, you'll be able to compile downloaded games whose source file **#Include**s any of those forms.

How do I compile on a PC running XP?

The PC version of the Inform compiler, `Inform.exe`, is a Windows *console* application; it doesn't have its own Graphical User Interface (GUI) with pull-down menus and dialog boxes, but instead expects to be given the information it needs (such as the name of the file you wish to compile) on the command line. This means that nothing much happens if you just double-click on its icon, since this runs the compiler but doesn't tell it what to compile. To be able to do that, you need to run the compiler in a Command Prompt window. Click **start** and then **Run...**, type **command** and press Enter (or alternatively click **start** and select **Programs**, **Accessories** and finally **Command Prompt**): a Command Prompt window appears. Make the folder (`Inform\Games\MyGame1`) containing your first game (`MyGame1.inf`) your 'current directory' by typing something like **cd \*path*\to\*game_folder*** and try to run the compiler:

```
  C:\> cd \My Documents\Inform\Games\MyGame1
  C:\My Documents\Inform\Games\MyGame1> Inform  MyGame1

  'Inform' is not recognized as an internal or external command,
  operable program or batch file.
```

Unfortunately, if you've followed our advice about keeping the compiler and library files separate from your games, XP won't be able to find the compiler, and so doesn't recognize your 'Inform' command. Instead, type this longer form which tells the computer where you've stored the compiler:

```
  C:\My Documents\Inform\Games\MyGame1> ..\..\Lib\Base\Inform  MyGame1

  Inform 6.30 for Win32 (27th Feb 2004)
  MyGame1.inf(7): Fatal error: Couldn't open source file "Parser.h"
```

This is better -- the compiler has started to run -- but there's still a problem: you now discover that the compiler can't find the Library files, so their location must also be specified:

```
  C:\My Documents\Inform\Games\MyGame1> ..\..\Lib\Base\Inform
+include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib  MyGame1

  Inform 6.30 for Win32 (27th Feb 2004)
```

That's it; the compilation has worked (really -- you don't see any 'successful' message). But it's a real pain having to open a Command window and type all that stuff each time you compile; it's much cleaner to put the commands in a batch file `MYGAME1.BAT`:

```
  ..\..\Lib\Base\Inform  +include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib
MyGame1
```

This is a file which you *can* double-click; it automatically opens a Command window and runs the compiler in it, but then immediately closes the window before you can see what happened. To prevent this auto-close, add a second line to your batch file:

```
  ..\..\Lib\Base\Inform +include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib
MyGame1
  pause "at end of compilation"
```

And now you can double-click `MYGAME1.BAT`, the compiler runs and you can read any error messages, before pressing any key to close the Command Prompt window. A better solution than all of this, however, is to get yourself a decent text editor within which you can both compile and test your work-in-progress -- see the section on IDEs.

In a few cases, it's useful to capture compiler information to a file; for example, when you use flags like **-u** (work out most useful abbreviations) or **-z** (print memory map of the Z-machine). Do this by adding to the end of the compilation command:

```
  ..\..\Lib\Base\Inform       +include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib
MyGame1  >MyFile.txt
```

Let's spend just a little longer on that +include_path= parameter. This is telling the compiler where to look for files which you mention in **Include** directives, for example when you say

`Include "Parser";`. The parameter specifies three places, separated by commas, for the compiler to try:

Parameter	Refers to
`.\`	The current folder, containing the game source file.
`..\..\Lib\Base`	Up one folder level (to Games), up another folder level (to Inform), then down again to the Lib folder and the Base folder below that.
`..\..\Lib\Contrib`	Up one folder level (to Games), up another folder level (to Inform), then down again to the Lib folder and the Contrib folder below that.

The compiler tries those three locations in sequence each time that it needs to **Include** a file into your game source: first the game's own folder, then the Base folder (where the standard Library files are stored), and finally the Contrib folder (where contributed extensions are stored). This scheme has a couple of advantages. First, it keeps the various file types nicely separated; this makes them easier to find, and helps to prevent confusion. Second, it means that you can make experimental changes to Library files and contributed files if you need to; just copy the file in question into the game's folder, and make your changes to that copy. Because the compiler looks first in the game's folder, it will use, for example, a copy of `Parser.h` from that folder in preference to the one in Lib\Base. You can experiment with changing the standard parser (if you're feeling brave) at no risk: other games won't be affected, and if you simply delete your modified form of the file, the compiler will revert to using the standard version.

How do I compile on a Mac running OS X?

The Inform compiler for OS X must be run through the Unix **Terminal** utility. If you double-click on its icon from a **Finder** window, nothing much happens: a Terminal window opens and runs the compiler without parameters, which triggers a "basic usage" message and ends up by stating that no compilation was requested. The compiler needs to be told which file to compile, and we must oblige by explicitely typing it.

There are two ways to approach this problem. One requires you to interact with the Unix Terminal for each compilation, while the other implies a bit of pre-configuring whenever you start a new game or project but then lets you operate through the Finder. In both cases, we're assuming that you're using the folder setup that was mentioned in the previous entry, and that the `Inform` folder is in your home directory.

Compiling with Terminal: part 1 -- the basics

We're using Mac OS X 10.3 (Panther) at the moment, which has three different Unix shells, `tcsh`, `bash` and `zsh`. By default Terminal launches the `tcsh` shell (despite the Mac Help telling you that "by default you use the `bash` shell"). This only becomes relevant a bit later on, when we'll tell you about a few syntax differences.

Go to **Applications/Utilities** and double-click on **Terminal**. This opens the utility which provides you with a set of windows to access the Unix command line. Supposing the computer is named `Computer`, and the user `Boojum`, you should see something like this:

Unix lore: A "shell" is a program that lets you interact with Unix, and it runs when you open a Terminal window. Its function is to interpret what you type on the keyboard, launch other programs (like cd or ls), display on the screen what those programs have to say and then await your pleasure in case there was something else you wanted. There are many different shells for Unix, each offering particular features and scripting languages.

If you're unfamiliar with the use of Unix on your Mac, there are a few interesting tutorials out there, like: Mac OS X Unix Tutorial

```
  Last login: Wed Jun 30 18:05:55 on ttyp1
  Welcome to Darwin!
  [Computer:~] Boojum%
```

You have now the ability to type Unix commands in this window. `cd` enables us to change the current working folder and `ls` lists its contents (remember that names in Unix are case-sensitive; "LS" is different from "ls"). You can find out which shell you're currently using by typing:

```
  echo $SHELL
```

When the Terminal utility starts, you are in your home directory, here denoted by the `~` symbol in `[Computer:~]`. In the folder `Inform/Games/MyGame1` we have included `MyGame1.inf` which is a tiny skeleton game in Inform source format. This is the file we're going to use in order to test Inform. Type:

```
  cd Inform/Games/MyGame1
```

We're now in the game folder. If you type `ls` you should see its contents:

```
  About.htm         MyGame1.command      MyGame1.inf
```

The current release of the Inform compiler resides in the `Inform/Lib/Base` folder, and it's called `inform630_macosx`. Bear in mind that the name of the file `inform630_macosx` may have changed if a new version of the compiler has been released and this FAQ entry isn't up to date. In that case substitute `inform630_macosx` with the name of the version that you're using.

We could now try to run the compiler from here by typing:

```
  ~/Inform/Lib/Base/inform630_macosx MyGame1
```

As we said before, you need to specify the source file that you want to compile, in this case `MyGame1.inf`, but we don't need to mention its extension. After entering the above command, we're presented with the following output:

```
  Inform 6.30 (27th Feb 2004)
  line 7: Fatal error: Couldn't open source file "Parser.h"
```

That "Fatal error" sounds worse than it really is. What we're seeing here is that the compiler has run, opened `MyGame1.inf`, and encountered a problem because in Line 7 of the source there was a reference to a library file that the compiler isn't able to find. We can improve our efforts by indicating to the compiler which folders should be investigated for interesting files (it's one long line, in this document divided because it doesn't fit):

```
  ~/Inform/Lib/Base/inform630_macosx
+include_path=./,../../Lib/Base,../../Lib/Contrib MyGame1
```

And now we should see only this (somewhat terse) output:

```
  Inform 6.30 (27th Feb 2004)
```

This is the compiler's way of saying that everything worked peachy, following the old-school philosophy that perfection means "nothing to complain about". The outcome can be seen if you type `ls` again. Among the files listed, you'll see a new one, `MyGame.z5`, which is the story file that can be played using a Z-machine interpreter.

Compiling with Terminal: part 2 -- adding an alias

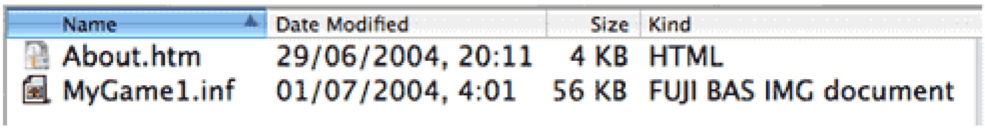
This basic method of compilation is rather clumsy and inconvenient, requiring a lot of typing. There are three parts in our longish command:

`inform630_macosx` refers to the compiler program, and `~/Inform/Lib/Base` is the name of the folder which contains it.

`+include_path=./,../../Lib/Base,../../Lib/Contrib` tells the compiler where to look for files like `Parser` and `VerbLib` which you've Included in the source file.

Three locations are suggested, separated by commas: this folder, which holds the source file (`./`); the folder holding the standard library files (`../../Lib/Base`); the folder holding useful bits and pieces contributed by the Inform community (`../../Lib/Contrib`). The three locations are searched in that order.

`MyGame1` is the name of the Inform source file that we want to turn into a story file.



Name	Date Modified	Size	Kind
About.htm	29/06/2004, 20:11	4 KB	HTML
MyGame1.inf	01/07/2004, 4:01	56 KB	FUJI BAS IMG document

By convention, all Inform source files have an extension of `.inf`. However, Mac OS X may show its Kind as "FUJI BAS IMG document" and attempt to open it with **GraphicConverter**. If you're not a regular user of FUJI BAS IMG documents, you may want to change this: right-click on the file (or Ctrl-click), select **Get Info**. In the **Open with** tab, select **TextEdit** as the application and press the **Change All...** button. You're asked for confirmation.

Regarding the first two parts, the `alias` command comes to our rescue:

```
  alias inform ~/Inform/Lib/Base/inform630_macosx
+include_path=./,../../Lib/Base,../../Lib/Contrib
```

This is telling the Unix shell that typing `inform` is equivalent to typing

`~/Inform/Lib/Base/inform630_macosx`
`+include_path=./,../../Lib/Base,../../Lib/Contrib`.

Now we have access to the compiler (and the compiler has access to the library files) through the single word `inform`, regardless of the current working folder.

> **Note**: The `alias` command is unfortunately one whose syntax isn't consistent in all Unix shells. If you're running `zsh` or `bash`, you have to type:
>
> ```
> alias inform='~/Inform/Lib/Base/inform630_macosx
> +include_path=./,../../Lib/Base,../../Lib/Contrib'
> ```

With the alias defined, you can type simply:

```
  inform MyGame1
```

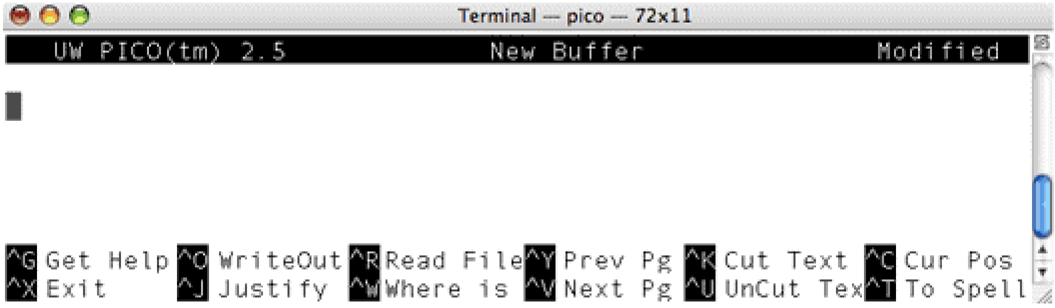
and, as before, you should see the compiler run and report no errors:

```
  Inform 6.30 (27th Feb 2004)
```

Compiling with Terminal: part 3 -- making the alias permanent

There's just one small glitch in this process. The `alias` that we have established is good only for this session. If you close the Terminal window and open a new one, the alias is gone. There's a fairly easy way to make it stick permanently: write it into a configuration file that is read every time that you open a Terminal window and launch the Unix shell.

To make the configuration file, we need a Unix text editor. The Unix shell in your OS X already has one, named **pico**. Open a new Terminal window (to ensure that we're at our home directory), type `pico` at the command prompt and you should see something like this:



```
   UW PICO(tm) 2.5                New Buffer                     Modified

^G Get Help ^O WriteOut ^R Read File^Y Prev Pg ^K Cut Text ^C Cur Pos
^X Exit     ^J Justify  ^W Where is ^V Next Pg ^U UnCut Tex^T To Spell
```

Type the `alias` line that we need:

```
  alias              inform             ~/Inform/Lib/Base/inform630_macosx
+include_path=./,../../Lib/Base,../../Lib/Contrib
```

Now press **Ctrl-O** to save the configuration file. You're prompted for a name, which has to be very specific for this to be understood as a configuration file (watch that initial period):

```
  .tcshrc
```

... and hit **Enter**. Now, press **Ctrl-X** to exit pico. You'll be back at the command prompt. If you ever need to change the settings of the configuration file (because you have moved the Inform folder to a different location or because you downloaded a new version of the compiler), open a Terminal window and type:

Unix tip: If you type the command **ls** in your home directory, you will see no **.tcshrc** file. This is because files beginning with a period are considered to be "hidden". You must use **ls -a** instead.

```
  pico ~/.tcshrc
```

Make the desired changes, save it with **Ctrl-O**, confirm the file name by pressing **Enter**, and exit with **Ctrl-X**.

> **Note**: The `.tcshrc` configuration file is good only for the `tcsh` shell. You can follow a similar process if you work with `bash` or `zsh`, but the name and location of the configuration files vary.

To verify that everything works as it should, open a new Terminal window (**Ctrl-N**). Now the configuration file should have been incorporated into the Unix shell. Go to the game folder:

```
  cd ~/Inform/Games/MyGame1
```

and type:

```
  inform MyGame1
```

You should see our laconic friend:

```
  Inform 6.30 (27th Feb 2004)
```

If, on the other hand, you get an error of some kind, something's amiss. Double-check these instructions, make sure that the path and filenames are correct and please confirm that the ever-important `.tcshrc` file is in your home directory.

Compiling with Finder: part 1 -- using a command-line file

If you have followed the above instructions to configure your system, every time that you need to compile your source code you must open a **Terminal** window, browse to the *`game_folder`* where your source file resides and type `inform` *`source_file`*. If you don't close the Terminal window, the next time you need to compile you only have to press the **up-arrow** key and the last command that you typed (`inform` *`source_file`*) re-appears, making compilation just a two-keystroke action.

We could avoid the need to open a Terminal window if we create an executable command-line file (also known as a "shell script"). Basically, you can create a text file which includes a bunch of Unix commands and make it executable, so that you may simply double-click on it from the Finder and forget about the hassle of using the Terminal utility while coding your game.

The download includes such an executable in `Inform/Games/MyGame1`, called `MyGame1.command`. This file contains two lines of instructions, editable with any text editor:

```
  cd ~/Inform/Games/MyGame1

  ../../Lib/Base/inform630_macosx
+include_path=./,../../Lib/Base,../../Lib/Contrib  MyGame1
```

The first one sets `~/Inform/Games/MyGame1/` as your working directory, and the second one compiles `MyGame1.inf`, as we have seen in the previous section.

To test that it works, move the story file `MyGame1.z5` to the Trash and then double-click `MyGame1.command`. A shell window opens to tell you about the process (and to list compilation errors when necessary). If the final lines look something like this:

```
  Inform 6.30 (27th Feb 2004)
  logout
  [Process completed]
```

... it means that compilation was successful. A new `MyGame1.z5` story file pops up in your folder.

You can copy this executable file to any folder where there is a source file to compile, but you'll have to make changes to the working folder path and the name of the source file.

Compiling with Finder: part 2 -- editing the command-line file

There are two peculiarities that lets your system understand that `MyGame1.command` is a Terminal Shell Script. Mac Help will tell you that "the `.command` filename extension is not required", whereas in fact it's essential. You also need to set an attribute of the file which marks it as "executable" (the "executable bits"). If it doesn't meet both conditions, `MyGame1.command` won't run as it should.

You have to be careful when editing this file: if you were, for instance, to open it in a text editor and save it to a different location with a different name, the executable bits might get lost, and when you double-click it, you would see an error message saying that the file could not be opened because it's probably not executable.

To make a command file from scratch (also, to fix the problem of "lost" executable bits) you can follow these simple steps:

Open any text editor and write (using your own values for the *game_folder* and *source_file*):

```
        cd ~/Inform/Games/game_folder

         ../../Lib/Base/inform630_macosx
       +include_path=./,../../Lib/Base,../../Lib/Contrib  source_file
```

Save the file in the folder *game_folder* and call it *source_file*`.command`. Make sure that the text editor doesn't append a `.txt` extension. If it does, rename the file manually.

Open a **Terminal** window and browse until you are in the folder *game_folder*.

Type:

```
        chmod 777 source_file.command
```

This command sets all the executable bits for the file and marks it as "executable".

Close the Terminal window.

Now, everytime you need to compile your game, you can just double-click on `source_file.command` from the **Finder**.

Running the game after compiling with Terminal or Finder

After a successful compilation, if you look at the Finder window of `MyGame1`, there should be a story file `MyGame1.z5`. Use the **Finder** to display the contents of the `Inform/Bin/Zoom` folder, and double-click **Zoom**, the game interpreter. It will present an **Open** dialog box. Browse to display the `Inform/Games/MyGame1` folder, and open `MyGame1.z5`.

When the system first "sees" the Zoom interpreter, it automatically creates an association with story files whose extension is `.z5` (and with other Infocom formats). From now on, you'll be able to play a game simply by double-clicking its story file.

What can I expect when I try to compile my program?

The short answer is: lots of errors. If you think of the compiler as a tool for finding your programming mistakes, with the generation of runnable Z-code as a fortuitous by-product, you'll be close to the mark. Just occasionally a compilation is successful, but that's the exception rather than the rule; it's much more likely that it will fail, probably producing a lot of obscure messages. So, you should be neither surprised nor worried by a mass of error reports.

When you see a long list of compilation errors, look only at the *first* one. Solve that problem and recompile: you'll often find that several of the error messages -- not only the first one -- disappear. So then concentrate on what's now become the first error, fix that... and repeat this cycle until the game compiles cleanly.

How do you know that you've got a clean -- error-free -- compilation? Because the compiler doesn't complain (it's like the dog that *didn't* bark in the night). If all that you see is something like this:

```
Inform 6.30 for Win32 (27th Feb 2004)
```

then things have gone well for you.

Every error message mentions the line number in your game where the error was detected, though that line isn't necessarily where the actual mistake occurs. Still, it's the best clue you'll get, so study that line, and the one before it, looking for things like: missing or misplaced commas and semicolons; mismatched pairs of apostrophes **'...'**, quotes **"..."**, parentheses **(...)**, brackets **[...]** and braces **{...}**; misspelled names for objects or variables.

As well as errors, you may also encounter warning messages. While these aren't as serious -- they don't prevent generation of Z-code -- it isn't a good idea to ignore them. A warning usually means that something isn't quite as you'd planned; you'll help yourself by fixing warnings as well as errors, so that your game compiles completely cleanly.

Here's an example which illustrates some of this. Suppose that you'd typed the simple example shown earlier in So, what *is* Inform?, making the understandable mistake of forgetting to put a semicolon at the end of line 19:

```
[ Initialise
```

The compilation produces these messages:

```
C:\My Documents\Inform\Games\Ruins\Ruins.inf(19): Error:  Expected local
variable name or ';' but found =
>         location =

C:\My Documents\Inform\Games\Ruins\Ruins.inf(19): Warning:  Local variable
"location" declared but not used

Compiled with 1 error and 1 warning (no output)
```

You'll see both an Error which (sort of) suggests that a semicolon may be missing, plus a completely spurious Warning about "location" not being used. Once you spot the missing semicolon, the Warning disappears also.

Why does my game start off so big?

The very smallest program using the Inform language is perhaps:

```
[ Main; "ok"; ];
```

and the compiled size of that is just 1.5Kb -- pretty compact, but not a whole bundle of fun as a piece of interactive fiction.

A minimal IF game needs at least one room, and *that* calls for the library files to be Included, so for all practical purposes your starting point is something more like this:

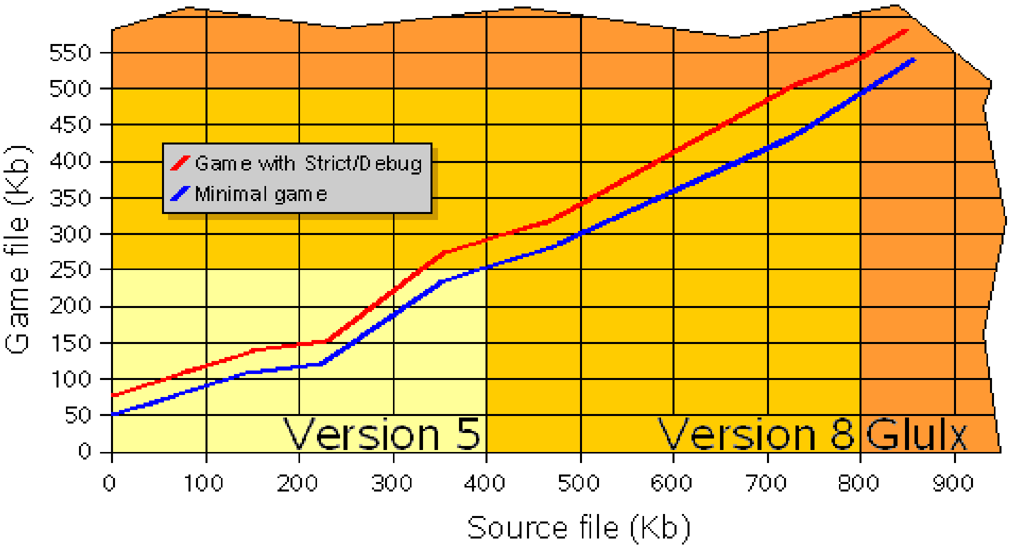
```
Constant Story "STORY";
Constant Headline "^HEADLINE^";

Include "Parser";
Include "VerbLib";

Object  theRoom "A room"
  with  description "Empty.",
  has   light;

[ Initialise;
    location = theRoom;
];

Include "Grammar";
```

which clocks in at a hefty 48.5Kb, of which only about 50 bytes is due to your strings, 'theRoom' object and 'Initialise' routine: literally 99.9% of that file is the parser, verb definitions, standard directions and other model world fundamentals provided by the library. So that's your realistic baseline: every game starts off with approximately 50Kb of standard -- and virtually indispensible -- overhead.

In fact, they start off as small as that only if you specify the **-~S** compiler switch, which deactivates Strict mode. If you specify **-D~S** then you get Debug (but not Strict) mode and the game grows to 55Kb, while **-DS** gives you both modes, at 76Kb. (For completeness, adding the **-X** switch incorporates the Infix debugger as well, for a total size of 137Kb; that isn't something you often need, though.) In fact, **-S** is the compiler's *default* setting, so even if you specify no switches at all, that's what you'll get. Fortunately, this isn't an issue: you *want* Strict (and usually Debug) modes to be active when you're authoring a new game, because of the assistance they give you in defeating silly programming errors.

We've talked about the *minimum* size of a game; what about the *maximum* size? Well, that's well-defined: it's 256Kb if you compile to Version 5, which is the default, and 512Kb if you use the **-v8** switch to compile to Version 8. That's double the size limit at almost no cost, which is a damn good deal. (It's not entirely free; the same game is about 2% bigger in Version 8 than in Version 5, because in the V8 format there's a little more space wasted between each string and each routine, but the loss is pretty trivial.) And if you supply the **-G** switch to compile for the Glulx virtual machine, then there's effectively no size limit at all.

Finally, is there any way of telling how big your game will be when finished? Only very roughly. Once you've written a reasonable proportion, say one third to one half, you should be able to guess at the eventual quantity of source code, and then use this chart to estimate the final size of the compiled game.



The chart shows source size (horizontally) versus compiled game size (vertically). The points are measured from real games, but you should treat this data with caution: lots of factors can influence the results, such as the quantity of comments in the source, and the number of library extensions which you Include into your game. The rule of thumb seems to be: up to 400Kb of source will compile to Version 5, up to 800Kb of source will compile to Version 8, and if you've got more source than that, you're probably looking at a Glulx game.

How do I change the compiler settings?

More information in the DM: §39

When you run the Inform compiler, you have as a minimum to tell it the name of the source file containing your game. For example, assuming that **inform** represents the compiler program, and that your current folder/directory contains an error-free source file **myGame.inf** as well as all of the library files, then on a PC this command should be sufficient to create a Z-Machine game **myGame.z5**:

```
  Inform myGame
```

Often, you need to provide additional information to the compiler. This information comes in four forms: **ICL files**, **switches**, **path variables** and **memory settings**, and can be provided using two methods: **on the command line** and **in the source file**. We'll describe the two methods first.

Method: On the command line

This is the traditional way on many platforms: you simply give the additional information before or after the name of your source file. For example:

```
  Inform -X myGame +include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib
```

This compiles **myGame.inf**, searching for Included files in the specified path, and incorporating the Infix debugger into **myGame.z5**.

Method: In the source file

Starting with Inform 6.30, you can also supply any additional information as special comment lines at the start of your source file. For example, if **myGame.inf** begins thus:

```
!% +include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib
!% -X
Constant Story "...";
Constant Headline "...";
```

then you can run the compiler like this to have exactly the same effect as the previous example:

```
Inform myGame
```

The special comments beginning with **!%** must be the very first lines in the source file.

You can combine the two methods: a good plan is to define **+include_path=** on the command line (since it's probably the same for all games that you compile), while specifying switches and memory settings at the start of the source file (since they're game-specific, and often need changing as your game develops).

There's also a **Switches** directive which enables some switches to be set within the source file, but it's less useful and has been superseded by the **!%** technique.

Info: ICL files

ICL files are given in parentheses '**(***filename***)**'. An Inform Command Language file can contain any of these four forms, one to a line. For example:

```
inform (myCommands) myGame
```

Info: Switches

More information in the DM:Table 3

Switches are introduced by a minus sign '**-**'. For example, **-v8** specifies Version 8 output, **-z** asks the compiler to print a Z-Machine memory map, and **-~S** turns off Strict mode. You can merge several switches behind a single minus; **-v8z~S** is the same as **-v8 -z -~S**. If you keep them separate, you need spaces between each (**-v8-z-~S** is wrong). Case matters: **-s** is different from **-S**. For example:

```
Inform -~Sv8 myGame
```

Info: Path variables

Path variables are introduced by a plus sign '**+**', and specify directories to be used by the compiler. The default in all cases is the current directory:

Variable	Specifies
+include_path=*dir*,*dir*,*dir*,...	One or more directories to be searched in sequence when handling a **Include** directive. This is probably the only path variable that you're likely to need.
+source_path=*dir*,*dir*,*dir*,...	One or more directories to be searched in sequence when looking for the source file.
+icl_path=*dir*,*dir*,*dir*,...	One or more directories to be searched in sequence when looking for an ICL file.
+code_path=*dir*	The directory where the compiler should write the Z-code game file.
+temporary_path=*dir*	The directory where the compiler should write any temporary files that it needs.

For example:

```
Inform myGame +include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib
```

Memory settings

Memory settings are introduced by a dollar sign '**$**', and control how much memory the compiler thinks it will need to compile the game. You need worry about these settings only if a compilation fails with a message that you need to increase one of the values:

Setting	Specifies
$MAX_ABBREVS=64	The maximum number of declared abbreviations. It is not allowed to exceed 64.
$MAX_ACTIONS=200	The maximum number of actions -- that is, routines such as TakeSub which are referenced in the grammar table.
$MAX_ADJECTIVES=50	The maximum number of different "adjectives" in the grammar table. Adjectives are misleadingly named: they are words such as "in", "under" and the like.
$NUM_ATTR_BYTES=6	The space used to store attribute flags. Each byte stores eight attribytes. In Z-code this is always 6 (only 4 are used in v3 games). In Glulx it can be any number which is a multiple of four, plus three.
$MAX_CLASSES=64	The maximum number of object classes which can be defined. This is cheap to increase.
$MAX_CLASS_TABLE_SIZE=1000	The number of bytes allocated to hold the table of properties to inherit from each class.
$MAX_DICT_ENTRIES=2000	The maximum number of words which can be entered into the game's dictionary. It costs 29 bytes to increase this by one.
$DICT_WORD_SIZE=6	The number of characters in a dictionary word. In Z-code this is always 6 (only 4 are used in v3 games). In Glulx it can be any number.
$MAX_EXPRESSION_NODES=100	The maximum number of nodes in the expression evaluator's storage for parse trees. In effect, it measures how complicated algebraic expressions are allowed to be. Increasing it by one costs about 80 bytes.
$MAX_GLOBAL_VARIABLES=240	The number of global variables allowed in the program. (Glulx only)
$HASH_TAB_SIZE=512	The size of the hash tables used for the heaviest symbols banks.
$MAX_INCLUSION_DEPTH=5	The number of nested Includes permitted.
$MAX_INDIV_PROP_TABLE_SIZE=15000	The number of bytes allocated to hold the table of ..variable values.
$MAX_LABELS=1000	The maximum number of label points in any one routine. (If the -k debugging information switch is set, MAX_LABELS is raised to a minimum level of 2000, as about twice the normal number of label points are needed to generate tables of how source code corresponds to positions in compiled code.)
$MAX_LINESPACE=16000	The size of workspace used to store grammar lines, so may need increasing in games with complex or extensive grammars.
$MAX_LINK_DATA_SIZE=2000	The size in bytes of a buffer to hold module link data before it's written into longer-term storage. 2000 bytes is plenty.
$MAX_LOCAL_VARIABLES=16	The number of local variables (including arguments) allowed in a procedure. (Glulx only)

$MAX_LOW_STRINGS=2048	The size in bytes of a buffer to hold all the compiled "low strings" which are to be written above the synonyms table in the Z-machine.
$MAX_NUM_STATIC_STRINGS=20000	The maximum number of compiled strings allowed in the program. (Glulx only)
$MAX_OBJECTS=640	The maximum number of objects. (If compiling a version-3 game, 255 is an absolute maximum in any event.)
$MAX_OBJ_PROP_COUNT=128	The maximum number of properties a single object can have. (Glulx only)
$MAX_OBJ_PROP_TABLE_SIZE=4096	The number of words allocated to hold a single object's properties. (Glulx only)
$MAX_PROP_TABLE_SIZE=30000	The number of bytes allocated to hold the properties table.
$MAX_QTEXT_SIZE=4000	The maximum length of a quoted string. Increasing by 1 costs 5 bytes (for lexical analysis memory). Inform automatically ensures that MAX_STATIC_STRINGS is at least twice the size of this.
$MAX_SYMBOLS=10000	The maximum number of symbols -- names of variables, objects, routines, the many internal Inform-generated names and so on.
$MAX_STATIC_DATA=10000	The size of an array of integers holding initial values for arrays and strings stored as ASCII inside the Z-machine. It should be at least 1024.
$MAX_STATIC_STRINGS=8000	The size in bytes of a buffer to hold compiled strings before they're written into longer-term storage. 2000 bytes is plenty, allowing string constants of up to about 3000 characters long. Inform automatically ensures that this is at least twice the size of MAX_QTEXT_SIZE, to be on the safe side.
$SYMBOLS_CHUNK_SIZE=5000	The symbols names are stored in memory which is allocated in chunks of size SYMBOLS_CHUNK_SIZE.
$MAX_TRANSCRIPT_SIZE=200000	Allocated only for the abbreviations optimisation switch, and has the size in bytes of a buffer to hold the entire text of the game being compiled: it has to be enormous, say 100000 to 200000.
$MAX_VERBS=200	The maximum number of verbs (such as "take") which can be defined, each with its own grammar. To increase it by one costs about 128 bytes. A full game will contain at least 100.
$MAX_VERBSPACE=4096	The size of workspace used to store verb words, so may need increasing in games with many synonyms: unlikely to exceed 4K.
$MAX_ZCODE_SIZE=20000	The size in bytes of a buffer to hold compiled code for a single routine. (It applies to both Z-code and Glulx, despite the name.) As a guide, the longest library routine is about 6500 bytes long in Z-code; about twice that in Glulx.
$SMALL $LARGE $HUGE	Pre-defined values for all the above settings, for small, medium and large games respectively. Since the default is $HUGE and you generally only encounter any of these variables when the compiler tells you that you need to increase one of them, you can ignore these options.

For example:

```
  Inform myGame $MAX_STATIC_DATA=20000
```

You might run into a problem with the memory settings when compiling under Unix, since Unix command shells usually parse the dollar sign as a shell variable substitution. You need to put a backslash before it:

```
  Inform myGame \$MAX_STATIC_DATA=20000
```

Or single-quote it:

```
  Inform myGame '$MAX_STATIC_DATA=20000'
```

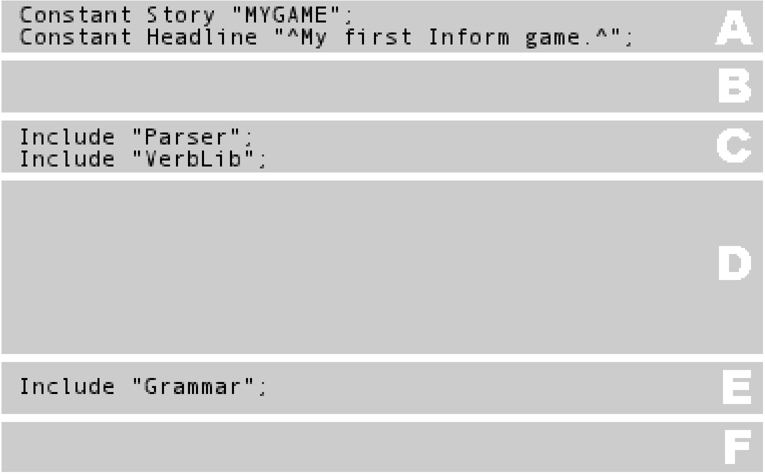
If you're using a Makefile, you have to go one step further, as the 'make' tool also treats the dollar sign as something special. In this case, something like this does the trick.

```
  myGame.z5: myGame.inf
      Inform '$$MAX_STATIC_DATA=20000' ...
```

Does it matter how I structure my game file?

Yes and no. The general structure of an Inform file always follows the same basic pattern. (If you want more background on the **#Include** files, see Where are all those Library files used?.)

More information in the DM: §4



```
Constant Story "MYGAME";
Constant Headline "^My first Inform game.^";

Include "Parser";
Include "VerbLib";

Include "Grammar";
```

Every game should contain those five lines shown in Areas A, C and E. Then, typically:

in Area B, define any Library constants (like **MAX_SCORE**) which configure the game's behaviour; this is also a good place for your own constants. If you are over-riding any Library routines, the appropriate **Replace** directive should go here.

in Area D, write the bulk of your code. The ordering of your objects, routines and global variables doesn't really matter -- except that globals must be defined before their first use -- so you can adopt any scheme that suits you.

in Area F, place any **Extend** and **Verb** directives, and any action routines for your new verbs.

if you are over-riding any Library messages, your **LibraryMessages** object should go in Area C between the **#Include**s of `Parser` and `VerbLib`.

if you are using the **#Include** directive to merge other people's Library packages into your game, the instructions with those packages will tell you where they should go. Usually, it's best to place them right alongside the three standard **Include**s in Areas C and/or E.

Does it matter how I organize my object definitions?

Yes and no. The general way of defining an object without a parent (for example, a room) is roughly:

```
Object  obj_name "external_name"
  with  prop_name value,
        prop_name value,
        ...
        prop_name value,
  has   attr_name attr_name ... attr_name;
```

and if there is a parent, one of:

```
Object  -> obj_name "external_name"
        ...

Object  obj_name "external_name" parent_obj_name
        ...
```

More information in the DM: §3.3 §4

A child object always has to be defined later in the file than its parent; if you're using the "->" method to denote parentage, you need to pay slightly more attention to the precise order of things than when you're using the *parent_obj_name* technique.

Within an object definition, you can put either the **with** segment first (as shown above) or the **has** segment; it's a matter of personal choice. Also, the ordering of properties within the **with** segment, and the ordering of attributes within the **has** segment, is down to you. The important thing is to select a style with which you feel comfortable, and then to apply it consistently.

To define an object which, as well as belonging to the generic class of Object, is also a member of a class which *you've* created (there's a bit more on classes later), the extended syntax is one of:

```
Object  obj_name "external_name"
  class class_name
  with  prop_name value,
        ...

class_name  obj_name "external_name"
  with  prop_name value,
        ...
```

More information in the DM: §3.8

Does it matter how I lay out my code?

Yes and no (sorry). That's a "no" because, as long as the code is syntactically correct, the compiler doesn't care in the slightest about its layout. That's a "yes" because, if you write your code to be human-readable, you help yourself to visualize its logical structure and anticipate how it will behave at run-time. The trick is: learn how to use whitespace -- that's blank lines and varying indentation -- so that you can see at a glance what's supposed to be going on.

One excellent way of handling indentation is to enforce a regular grid by using tabs. Any good text editor will allow you to specify a value for tab spacing: four is a good number, but two, three and eight are also popular settings. Just find a value which seems right to you, and then stick to it.

Stick also to a consistent way of wrapping braces **{...}** around blocks of code. Which standard you adopt isn't nearly as important as employing that standard across the board. You want the code to 'look right' to you.

And to other people. It's fairly common, when fighting an intransigent problem, to post code fragments on the Usenet newsgroup rec.arts.int-fiction, so that other Inform programmers can help with the debugging. You'll find a more willing response if your code has obviously been laid out with care.

Format it properly as you go. It's always a mistake to type the code in roughly, telling yourself that you'll come back later to tidy it up; you almost certainly won't. Enforce your own discipline of doing it properly first time, every time.

Roger Firth has a tool called INSTRUCTOR for cleaning up badly-formatted Inform code

Finally, when writing new code or changing an existing program, check frequently that the game still compiles cleanly. That way, you’ll catch silly syntax errors while your intentions are fresh in your mind.

How does a game begin?

A typical Inform game begins something like this; there are roughly three sections:

```
They thought that taking away your family would shatter your
will. They thought that feeding you rubbish would make you
plump, weak and lazy. They thought that wooden doors and metal
bars would be able to keep you in your place, until the hour
of doom.

They thought wrong.
```

The **Prologue** (optional) -- an opening text which normally sets up the mood of the narration and offers the first details of the story.

```
GROINK RETRIBUTION
An Interactive Porcine Vengeance.
Copyright (c) 2003 by Superb Author.
Release 1 / Serial number 030816 / Inform v6.21 Library 6/10
```

The **Banner** (optional), which displays the game's general information -- title, author, serial number, library version, etc.

```
In the pigsty
Mud, mud and more mud, enclosed by a wooden fence and one
stone wall to the north.

>
```

The description of the starting room, followed by the command prompt.

Let's take a look at the source code:

```
Constant Story "GROINK RETRIBUTION";
Constant Headline "^An Interactive Porcine Vengeance.
                   ^Copyright (c) 2003 by Superb Author.^";

Include "Parser";
Include "VerbLib";

Object  pigsty "In the pigsty"
  with  description
            "Mud, mud and more mud, enclosed by a wooden
             fence and one stone wall to the north.",
  has   light;

[ Initialise;
    location = pigsty;
    "^^They thought that taking away your family would shatter
     your will. They thought that feeding you rubbish would
     make you plump, weak and lazy. They thought that wooden
     doors and metal bars would be able to keep you in your
     place, until the hour of doom.
     ^^
     They thought wrong.^^";
];

Include "Grammar";
```

More information in the DM: §1.2

The DM tells us early on that all Inform programs must begin with the definition of a routine called **Main()** -- for a superficial overview of how a game runs, read this later topic -- which may invite new readers to believe that their game's source code must contain such a definition. That is not so. The library, which we Include in our source file (through `Parser`, `VerbLib` and `Grammar`), has already taken care of writing **Main()** and paving our way with necessary defaults.

We do need, however, to define a routine called **Initialise**, which runs before any text is printed and whose only mandatory task is to set the **location** variable to the starting room:

```
  location = pigsty;
```

Bear in mind that the **location** doesn't have to be a room: it may be a container (the player starts the game locked up inside a wardrobe) or a supporter (a few games start with the Player Character (PC) waking up on his bed).

Other than that, you can use **Initialise()** to write, set up and trigger just about anything that needs to be, well, *initialised*, before the game runs: assign starting values to variables, give possessions to the PC, start up daemons and timers which need to be functioning from the beginning, etc. In our example, we wish to display the prologue text, and we do it at the end of the routine with the quoted string statement -- which equates to (a) display the string, (b) output a newline character and (c) return.

More information in the DM: §21

Eventually, **Initialise()** runs out of statements and returns, at which moment the game banner is automatically printed. The banner is made up from three parts: the game title, defined by the constant **Story** (optional); the author's name, copyright info, description of game or whatever you wish for a headline, defined by the constant **Headline** (optional); and a line comprising the release number (1 by default, unless you define otherwise with the directive **Release**), the serial number (the date of compilation in YYMMDD format by default, unless you define otherwise with the directive **Serial**) and the versions of the compiler and the library. If you do not define the constants **Story** and **Headline**, the banner will consist only of this last line.

The banner is to players like the opening titles in a movie: from here on we're deep in the story/game business. However, you may want your game not to print a banner at the very beginning, because you wish, for example, to turn the prologue into an interactive opening sequence instead of a bunch of read-only paragraphs. For this, you must explicitly `return 2` at the end of **Initialise()**:

```
  [ Initialise;
      location = pigsty;
      print "^^They thought ...       ! explicit print statement doesn't return
             They thought wrong.^^"; ! after displaying the text.
      return 2;
  ];
```

This will stop the banner from appearing after **Initialise()** has run its course. Now, you should include the call

```
  Banner();
```

wherever it's appropriate for your game to print one; the copyright notice at the start of the DM explains the importance of the information it displays. Players, from their end, can summon up the banner whenever they want with the use of the VERSION command.

When **Initialise()** finishes, banner or no banner, the starting room comes up on the screen. That means that you must always define at least this room (in our example case, the pigsty). So let's take another look at our (slightly revised) source code:

This source code...	... means...	... and is
Constant Story "GROINK RETRIBUTION";	Defines the title of the game. (*)	Optional
Constant Headline "^An Interactive Porcine Vengeance. ^Copyright (c) 2003 by Superb Author.^";	Copyright info, description, etc. (*)	Optional
Release 1;	Game release (1 by default). (*)	Optional
Serial 030816;	Serial number (YYMMDD of compilation by default). (*)	Optional
Include "Parser";	Includes library file **Parser.h**	Mandatory
Include "VerbLib";	Includes library file **Verblib.h**	Mandatory
Object pigsty "In the pigsty" with description "Mud, mud and more mud, enclosed by a wooden fence and one stone wall to the north.", has light;	Definition of the starting room.	Mandatory
[Initialise;	Starting definition of the **Initialise()** routine.	Mandatory
location = pigsty;	Set **location** variable to starting room.	Mandatory
print "^^They thought that taking away your family would shatter your will... ^^They thought wrong.^^";	Prologue text.	Optional
return 2;	If present, the banner won't appear.	Optional
];	End marker of the **Initialise()** routine.	Mandatory
Include "Grammar";	Include library file **Grammar.h**	Mandatory

(*) These entries are part of the game banner.

Initialise() lets you effectively decide how to present your game to players, a delicate moment in which you seldom want to give the wrong impression. Your imagination is the only limit here, but let's take inventory of some examples and common uses.

More information in the DM: §21 §42 and in Marnie Parker's Stupid Initialise tricks

Ability to pause execution until the player presses a key

If you decide to write a long introduction, you can simply code a **print** statement, open the double quotation marks (") and begin typing your magnum opus until smoke comes out from the keyboard:

```
  print "Call me Ishmael. Some years ago...
         ...only found another orphan.
         ^^
         finis.";
```

At run-time, this procedure will fill the interpreter's screen with text and, when it reaches the bottom, a [MORE] prompt will appear, waiting for the player to hit a key in order to fill another screen. You may want, however, to have a little more control on the quantity of displayed text, in favour of suspense or to set the pace of information disclosure.

```
print "^Welcome to the Dark Ribbon Club. There
       are three ways to acquire membership:";
@read_char 1 i;                         ! Wait for the player to hit a key.
print "^^(a) Read Muffungus incantations to
       an audience of midget Praetorians
       under a moonless sky.";
@read_char 1 i;                         ! Wait for the player to hit a key.
print "^(b) Eat snails and oysters in rapid
       succession for seven days without pause.";
@read_char 1 i;                         ! Wait for the player to hit a key.
print "^(c) Become the slave of a club member
       until he decides to set you free.
       ^^
       The choice is all yours...";
```

The **@read_char** opcode is responsible for the trick. In this simple form, it pauses the game's execution until the player hits any key. Since it's using a variable (i), don't forget to declare it in the header of **Initialise()**:

```
[ Initialise i;
    ...
```

This technique may be combined with clearing the screen before the new text appears:

```
print "^Welcome to the Dark Ribbon Club.";
@read_char 1 i;                         ! Wait for the player to hit a key.
@erase_window -1;                       ! Clear screen.
print "^^The place where power and secrecy
       join forces to create a paroxysm of
       political greatness.";
```

Happily, version 6/11 of the Library makes this a little simpler. You can replace the **@read_char 1 i;** by **KeyCharPrimitive();** (no need to declare the variable **i**), and the **@erase_window -1** by **ClearScreen();**. Not only easier to remember, but also more powerful: the routines work for both Z-machine and Glulx.

Adding your own line to the banner

A common feature is a line displayed at the start of the game, suggesting that players might want to seek help, or read the game credits, or check the terms of the game's licence; a good place to do this is right after the banner information appears. So, call **Banner()** *within* **Initialise()**, then display your additional line, and then return 2 as we have seen above:

```
[ Initialise;
    location = pigsty;
    print "They thought...";  ! Display the prologue text.
    Banner();                 ! Call up the standard banner.
    print "Type HELP for information on licencing and credits,
           or if you get stuck.^";
    return 2;                 ! No need to display the banner again.
];
```

Calling up cool banners

If the regular banner does not satisfy your requirements and you need to produce a title page of astounding singularity, you should return 2 from **Initialise()** and make a call to your own printing routine:

```
[ Initialise i;
    location = pigsty;
    print "They thought...";  ! Display the prologue text.
    @read_char 1 i;           ! Wait for the player to hit a key.
    MyCoolBanner();           ! Call up your customised banner.
    return 2;
];
```

And then, of course, you must write a routine to suit your needs:

```
[ MyCoolBanner;
    ...
    your code here
    ...
];
```

This usually involves messing around with centring text, displaying boxed quotations or flashy effects like typing one character at a time, and other advanced stuff which is better left alone for the time being.

Changing defaults for library pre-defined objects or behaviour

Inform's library defines a whole bunch of objects and default configurations which are ready to use by the game designer. There are times when you might wish to alter some pre-defined characteristic. The description of the dark ("It is pitch dark, and you can't see a thing.") or that of the Player Character ("As good-looking as ever.") are common examples:

```
thedark.description = "You are engulfed in creepy shadows.";
player.description = "A slender youth in the prime of life, hungry for
                      experience and adventure.";
```

Or you may want to change the default inventory style:

```
inventory_style = FULLINV_BIT + ENGLISH_BIT + RECURSE_BIT;
```

Equipping the player with possessions

If the PC needs to start the game carrying some objects around, **Initialise()** is the perfect place to do it:

```
move axe to player;
move pants to player;
give pants worn;
```

The **move** statement makes the desired object a possession of the player. We are also setting the **worn** attribute so that the pants object comes into the scene covering all the embarrassing bits as opposed to being carried in the player's hands.

Changing the player into a user-defined character

To learn about the technical aspects of the player object defined by the library, you may consult this later topic. For now, let's just suppose that the human being with the pre-defined defaults represented by the library's **selfobj** are not suitable for your protagonist. We have already seen that it's easy to change its description -- or, in fact, other properties -- through the **player** variable, but if the changes are more extensive, you might need to define an altogether different player object:

```
Object  SelfVampire "(self object)"
  with  short_name [; return L__M(##Miscellany, 18); ],
        description  "As terrifying as ever.",
        ...
  has   animate concealed proper transparent;
```

In **Initialise()** you have two ways of making the change; through the variable **player**:

```
player = SelfVampire;
```

or with the library routine **ChangePlayer()**:

```
ChangePlayer(SelfVampire);
```

You should use either technique *before* you set the **location** variable. If you need to change the player again during the course of the game, you should use only **ChangePlayer()**.

Starting up timers and daemons

More information in the DM: §20

If a timer or a daemon (to learn more about them you may consult this later topic) needs to be functioning from the beginning of the game, you can easily activate them in **Initialise()** by calling their respective library routines:

```
  StartDaemon(CleaningRobot);
  SetTimer(VaultDoor, 10);
```

Setting up a game with "real" time

More information in the DM: §20

Every time that the player performs an action (except those defined as *meta* actions -- SAVE, RESTORE, RESTART and the like) it counts as one "turn". In the status line of a normal Inform game, we can see the current turn count right next to the current score. However, you can replace the turn and score counters by a clock displaying the time that is passing in the model world.

First, add this line at the top of your source code:

```
  Statusline time;
```

And then, in **Initialise()**:

```
  SetTime(time, rate);
```

where *time* is a number between 0 and 1439 defining the minutes since midnight, and *rate* is another number which specifies how many minutes you wish to push the clock forward at each turn. Suppose we want to start the game at 1:05 a.m., and each turn to count as five minutes:

```
  SetTime(65, 5);
```

Change the default BRIEF lookmode into SUPERBRIEF or VERBOSE

Normally, an Inform game will set BRIEF as the default look mode, which gives long descriptions of places never before visited and short descriptions otherwise. You can change this by setting the variable **lookmode** to a suitable value:

```
  lookmode = 1;              ! BRIEF mode.
  lookmode = 2;              ! VERBOSE mode (always long descriptions).
  lookmode = 3;              ! SUPERBRIEF mode (always short descriptions).
```

Ability to restore a previously saved game before anything interesting happens

Some games include lengthy prologues or introductory sequences and, as much as this narration technique might be necessary to give players all the relevant information before the action begins, it gets annoying to have to go through the same ramble every time you run the game. You can offer players the option to immediately restore a previously-SAVEd game:

```
  print "Would you like to restore an old game? ";
  if (YesOrNo()) <Restore>;
```

The **YesOrNo()** library routine waits for the player to type either YES (which makes the `if (...)` test **true** and executes the **Restore** action) or NO (which equals to **false**; execution proceeds without restoring).

How does a game end?

Every turn, Inform checks the value of a library variable called **deadflag**. It can be:

More information in the DM: §21

0 The usual state. Inform assumes that nothing drastic has happened to the Player Character (PC) and continues game execution.

1 Game has been lost -- the PC has died.

2 Game has been won.

3,4,... Game has been lost -- the PC has suffered a user-defined demise.

Note: It's remarkable the lack of apparent symmetry between Winning and Dying as the only pre-defined defaults of the library. It probably pays homage to Ye Olde Adventurers, who knew not of the word Losing, and nothing short of death would stop them in their tracks.

A value of **deadflag** other than zero will finish the game. To achieve this, you just have to make the assignment:

```
deadflag = 2;
```

wherever you need it, and the result will be:

```
    *** You have won ***


In that game you scored 35 out of a possible 100, in 325 turns.

Would you like to RESTART, RESTORE a saved game or QUIT?
```

We have three lines: (a) an informative message of the fate of the PC, in bold face and highlighted by asterisks to ensure that the point gets home; (b) a summary of the achieved score and turn count; (c) available options now that the game is over. The value of **deadflag** affects the demise-message -- though it does not affect the bold face or the asterisks. A value of 1 will print "You have died" and, for whatever other alternate endings you may need, you will be using values of 3 or more. In this case, you must provide a library entry-point routine called **DeathMessage()**, in which you specify the equivalence between **deadflag** values and final one-liners:

```
[ DeathMessage;
    switch (deadflag) {
      3: print "You acted unwisely.";
      4: print "You have shot your own foot.";
      5: print "You rot in jail.";
      ...
    }
];
```

You can customise a little the score line if you use the **PrintRank()** entry point routine, which lets you award flamboyant ranks to the player depending on the value of the **score** variable:

```
[ PrintRank;
    print ", earning you the rank of ";
    if (score >= 90) "Legend of the Far West.";
    if (score >= 70) "Marshall.";
    if (score >= 50) "Deputy.";
    if (score >= 30) "Cowboy.";
    "Tenderfoot.";
];
```

More information in the DM: §22

This will result in something like:

```
In that game you scored 35 out of a possible 100, in 325 turns,
earning you the rank of Cowboy.
```

The final Win/Lose line tells the player that he can now perform only three actions, RESTART, RESTORE a saved game or QUIT. This is not entirely accurate. If the game designer implements a score system based on tasks, players may at this point demand a FULLSCORE, which will tell them in detail how many points they got for each accomplished task. Additionally, players may invoke the command UNDO, thus returning to the previous turn, just before they managed to bring the game to one of its conclusions. You can make players aware of this possibility by writing the following constant:

```
Constant DEATH_MENTION_UNDO;
```

and the last line will change to:

```
Would you like to RESTART, RESTORE a saved game, UNDO your last
move or QUIT?
```

It is not uncommon for a game to include Easter-eggs or funny responses for highly improbable actions. These tend to go unnoticed by most players, but Inform gives you an ace up your sleeve to prevent the waste of your most inspired moments. You can define:

```
Constant AMUSING_PROVIDED;
```

along with an entry point routine called **Amusing()**, in which you can list all the interesting things that the player may have missed:

```
[ Amusing;
    print "Have you tried to kiss the Ancient Mariner?^
           Or place the pearl inside Coleridge's wig?^
           ...";
];
```

and, only if the player has won the game (**deadflag**=2), this will result in:

```
Would you like to RESTART, RESTORE a saved game, see some
suggestions for AMUSING things to do or QUIT?
```

Is there a good Integrated Development Environment?

As we speak, there isn't an Inform IDE which is generally agreed to be both reliable and effective, and which is in widespread use (though there are a few upcoming possibilities; see Mike Perlini's IF-IDE on the PC, and Scott Forbes' Yonk on the Mac). Until that situation changes, we survive by using a ordinary text editor like NotePad (not a word processor -- you don't want any funny formatting characters mixed in with the program).

Having said that, NotePad is not very powerful; an industrial-strength editor will do wonders for your Inform productivity, especially if it offers syntax colouring and the ability to compile and run Inform games without leaving the editor. Several good possibilities are listed in Roger Firth's List of IF Editors, and in the RAIF FAQ. For the PC, TextPad is my recommendation, and here are instructions on how to harmonize it with Inform. If you do this in conjunction with the folder structure suggested in the first topic, use this line when you come to the part about setting the Parameters for the Infrmw32 compiler:

```
$File -S +include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib
```

How do I use Modules?

Don't; they were designed to solve a problem which no longer exists.

The idea behind pre-compiled library modules was to cut down on the time taken to compile your game, back in the days when PCs were steam-powered and slow. Nowadays, PCs are much much faster, and the compilation time saving is tiny, much less than the period it would take you to learn how the module system is meant to work. Also, there are some snags to be aware of: various techniques for modifying the standard Library behaviour simply don't work as expected with pre-compiled modules; they aren't compatible with Version 8 stories, and so on. Most worryingly, there's the chicken and egg effect: because the module system is so little used, it is suspected of being more buggy than the rest of Inform, so people avoid using it, so the bugs aren't found... Be warned; modules simply aren't worth bothering about.

More information in the DM: §38

Can I write a game in French?

Yes, you can -- and in several other languages as wel (assuming, of course, that you already speak the language fluently). Inform is currently unique in being the only IF development system to offer this degree of support for non-English game creation. To write a game in one of these languages is simple; just do this:

More information in the DM: §34 §35 §36 §37

1. **Download two library files**. You must obtain replacements for the two standard library files which contain English-language text: the language definition file `English.h` (which defines parser-significant words like AND, EXCEPT and THEN; pronouns like IT and THEM; compass directions; small numbers; and all of the messages), and the file of basic verb definitions `Grammar.h`. The replacements are typically called, for example, `French.h` and `FrenchG.h`. Put these replacement files in the same folder as the other library files; there's no need to get rid of `English.h` and `Grammar.h`.

2. **Change one of the standard Includes**. Every English game source contains these three lines somewhere; the first two are fine as they are, but the third line should be changed to instead refer to the new grammar file `FrenchG.h`:

```
Include "Parser";
Include "VerbLib";
Include "Grammar";     ! change this line
```

3. **Modify your compilation process**. You control the operation of the compiler by setting command-line switches (for example **-X** or **-~S**) and variables (for example, **`+include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib`**).

   To compile using a non-English language, you need to defined another variable, thus: **`+language_name=French`**. The value that you provide here is used within `parserm.h` to Include the appropriate language definition file -- `French.h` in our example -- instead of the default `English.h`. Here's an example of a Windows batch file that could be used to compile a French version of RUINS.INF:

```
..\..\Lib\Base\Infrmw32 RUINS -S +language_name=French
+include_path=.\,..\..\Lib\Base,..\..\Lib\Contrib | more
pause "at end of compilation"
```

4. **Think about fonts**. If you intend to write in a language which makes heavy use of accented or unusual characters (such as Cyrillic or Greek), you'll also need to study the topic on character sets. With the Western European languages, this isn't usually a problem.

Rien de plus simple!

3 · Learning The Lingo

When are upper and lower case differentiated?

Mostly, they're treated the same. When you assign a name to something like an object or a routine, you can use either upper and lower case letters (you can also use underscores (_), and you can have digits anywhere except as the first character).

The difference between upper and lower case is important for:

- statements -- words like **if**, **print**, and **return** must be in lower case;
- strings -- text within quotes **"..."** respects upper and lower case differences;
- single-character constants -- for example, **'e'** and **'E'** have different values;
- print rules -- **(The)** and **(the)** are different rules, as are **(A)** and **(a)**;
- escape sequences -- for example **@^e** and **@^E** produce different output.

Even where case doesn't actually matter, there are a few conventions:

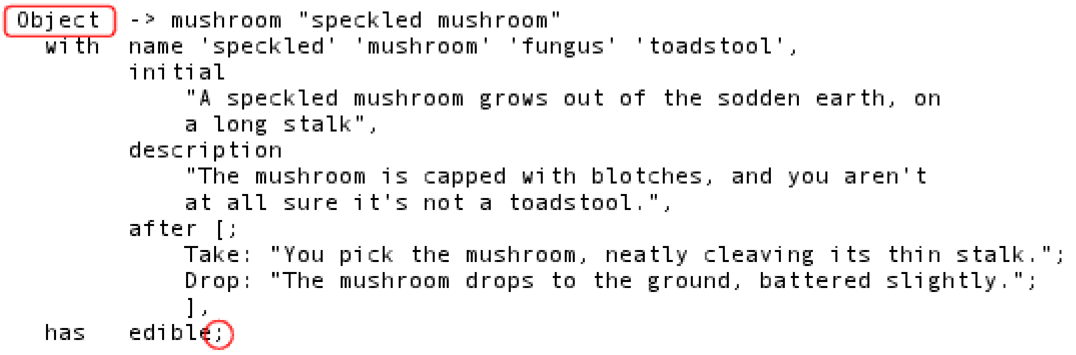
- dictionary words -- for example **'grumpy' 'old' 'man'**, are usually typed in lower case, since that's how they're stored anyway;
- directives like **Object** are given an initial capital letter;
- classes -- for example, a **Room** class -- are given an initial capital letter;
- actions -- for example a **Smile** action and a matching **SmileSub** routine -- have an initial capital;
- constants are often named in upper case, to help distinguish them from variables.

When do I use commas (,) and semicolons (;)?

You use a semicolon often; it's a *terminator* -- every statement and every directive ends with one. The comma is a *separator*, and you use it in only a few situations:

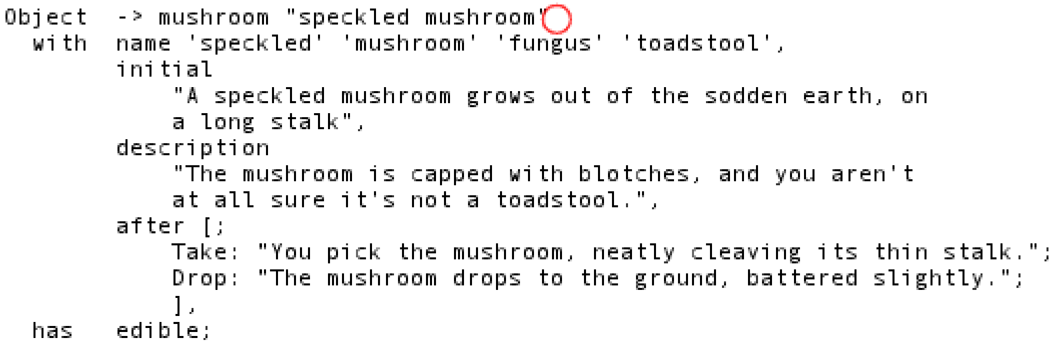
- between the properties of an object declaration;
- between the terms of a **print** statement;
- between the arguments of a call to a routine;
- between equivalent case values in a **switch** statement.

It's the first of those which seems to give novices most difficulty, so let's walk slowly through an example. This declaration starts with the **Object** directive, and ends several lines later with a semicolon:



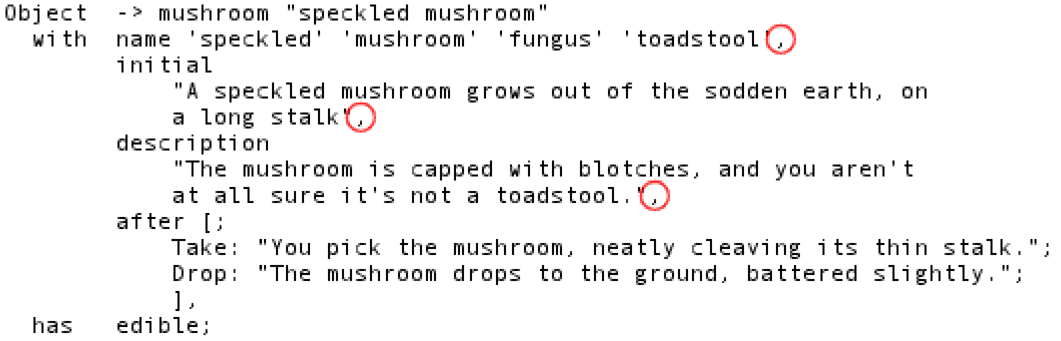
```
Object  -> mushroom "speckled mushroom"
  with  name 'speckled' 'mushroom' 'fungus' 'toadstool',
        initial
            "A speckled mushroom grows out of the sodden earth, on
            a long stalk",
        description
            "The mushroom is capped with blotches, and you aren't
            at all sure it's not a toadstool.",
        after [;
            Take: "You pick the mushroom, neatly cleaving its thin stalk.";
            Drop: "The mushroom drops to the ground, battered slightly.";
            ],
  has   edible;
```

You never need any punctuation in the object's header -- the first line which defines its internal and external names, and its parentage:



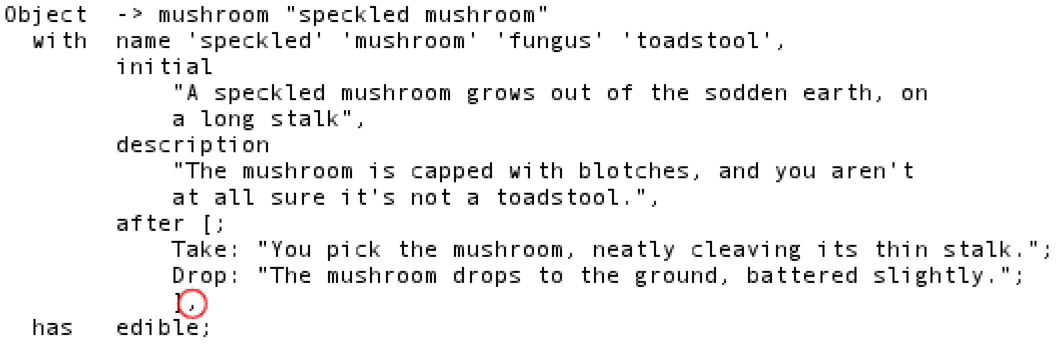
```
Object  -> mushroom "speckled mushroom"
  with  name 'speckled' 'mushroom' 'fungus' 'toadstool',
        initial
            "A speckled mushroom grows out of the sodden earth, on
            a long stalk",
        description
            "The mushroom is capped with blotches, and you aren't
            at all sure it's not a toadstool.",
        after [;
            Take: "You pick the mushroom, neatly cleaving its thin stalk.";
            Drop: "The mushroom drops to the ground, battered slightly.";
            ],
  has   edible;
```

You always need these commas, to separate the **name** property from the **initial** property, **initial** from **description**, and **description** from **after**:



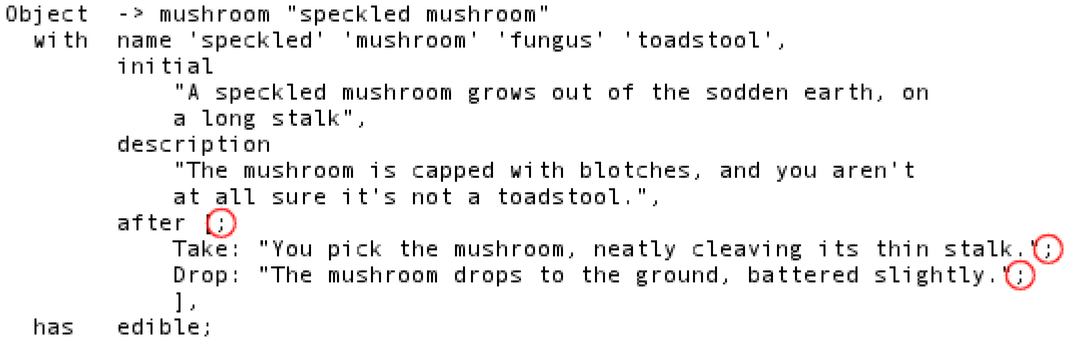
```
Object  -> mushroom "speckled mushroom"
  with  name 'speckled' 'mushroom' 'fungus' 'toadstool',
        initial
            "A speckled mushroom grows out of the sodden earth, on
            a long stalk",
        description
            "The mushroom is capped with blotches, and you aren't
            at all sure it's not a toadstool.",
        after [;
            Take: "You pick the mushroom, neatly cleaving its thin stalk.";
            Drop: "The mushroom drops to the ground, battered slightly.";
            ],
  has   edible;
```

A comma at the end of the final property definition, before the **has** segment, is optional. It's not needed, but it doesn't do any harm (and it makes things a bit easier if you later add another property to the list):



```
Object  -> mushroom "speckled mushroom"
  with  name 'speckled' 'mushroom' 'fungus' 'toadstool',
        initial
            "A speckled mushroom grows out of the sodden earth, on
            a long stalk",
        description
            "The mushroom is capped with blotches, and you aren't
            at all sure it's not a toadstool.",
        after [;
            Take: "You pick the mushroom, neatly cleaving its thin stalk.";
            Drop: "The mushroom drops to the ground, battered slightly.";
            ],
  has   edible;
```

These semicolons are within the **[...]** which surround the embedded property routine, and so have no effect on the structure of the mushroom's definition:



```
Object  -> mushroom "speckled mushroom"
  with  name 'speckled' 'mushroom' 'fungus' 'toadstool',
        initial
            "A speckled mushroom grows out of the sodden earth, on
            a long stalk",
        description
            "The mushroom is capped with blotches, and you aren't
            at all sure it's not a toadstool.",
        after [;
            Take: "You pick the mushroom, neatly cleaving its thin stalk.";
            Drop: "The mushroom drops to the ground, battered slightly.";
            ],
  has   edible;
```

When do I use apostrophes (') and quotes (")?

If your game mentions words in apostrophes (for example, **'grumpy' 'old' 'man'**), then the compiler stores those words in what's known as the game's dictionary. If your game mentions words or phrases in quotes (for example, **"grumpy old man"**) then you're defining a string, which the compiler stores in another place entirely. There's a really important distinction here: words in the dictionary are effectively *input* tokens, to be matched against words that the player types during the game. Strings are *output* messages, to be printed by your program during the game. For example, in this fragment:

```
>EXAMINE THE GRUMPY OLD MAN
The grumpy old man stands silently in the corner.
```

the game is comparing the player's input against the words that it knows (the contents of the dictionary), recognising an EXAMINE operation on an object identified by the tokens GRUMPY, OLD and MAN, and printing the description associated with that object. The object itself might have been defined thus:

```
Object  old_man "grumpy old man"
  with  name 'man' 'grumpy' 'cross' 'old' 'elderly' 'ancient' 'gnome',
        description "The grumpy old man stands silently in the corner.",
        ...
  has   animate;
```

and you can easily see the difference between the list of apostrophe-enclosed input tokens in the **name** property, and the quote-enclosed output string in the **description** property.

Unfortunately, Inform muddies this clear distinction by treating the **name** property (also the **Extend** and **Verb** directives) as a special case, and permitted you to enclose the input token in either apostrophes or in quotes. Please, don't take advantage of this unnecessary concession; stick to the simple and consistent policy: apostrophes around input (dictionary) words, quotes around output strings. Do that everywhere, and you won't get confused.

What does a string "..." as a statement by itself mean?

One of Inform's more unusual program statements is nothing more than a string in quotes:

```
"The grumpy old man takes no notice.";
```

which to the uninitiated can be pretty confusing. Just remember that this is merely a shorthand form of:

```
print_ret "The grumpy old man takes no notice.";
```

More information in the DM: §1.12

and that in turn these are both exactly equivalent to the three statements:

```
print "The grumpy old man takes no notice.";
new_line;
return true;
```

Once you grasp what's going on, you'll find this a useful and convenient shortcut, especially after an if statement; most people find the first of these two forms clearer to work with:

```
if (...) "The grumpy old man takes no notice.";
if (...) {
       print "The grumpy old man takes no notice.";
       new_line;
       return true;
       }
```

One aspect of the **"...";** statement that regularly fools newcomers is its inbuilt **return**. The statement outputs some text, and then *returns immediately* from whatever routine it occurs in, which of course means that any statement(s) following it cannot possibly be executed. Misunderstanding this behaviour is far and away the most likely cause of a "This statement can never be reached" compiler warning.

By the way, if you assign a string to a variable:

```
Global X = "The grumpy old man takes no notice.";
```

then you can achieve the same effect as a **"..."** statement with any of these:

```
print_ret (string) X;
print (string) X, "^"; return true;
print (string) X; new_line; return true;
```

What are the circumflex (^) and tilde (~) characters used for?

Inform uses both of these characters, which occur only rarely in English texts, for special purposes.

The *circumflex* can be included as part of a string within quotes "..." in order to output a newline character. For example:

More information in the DM: §1.11

```
print "^^^A line on its own.^^^";
```

prints three newlines before and after the line of text. If you wish to output a literal circumflex, you need to use the rather clumsy construct @@94, thus:

```
print "A line with a circumflex @@94 in the middle.^";
```

The circumflex can also be included in a dictionary word within apostrophes '...' in order to store a literal apostrophe as part of that word. For example:

More information in the DM: §1.4

```
name 'child^s' 'bicycle' 'bike' 'small',
```

might be part of the definition of an object which would respond to TAKE THE CHILD'S BICYCLE.

The *tilde* can be included as part of a string within quotes "..." in order to output a literal quote character. For example:

More information in the DM: §1.11

```
print "The bottle is labelled ~POISON~ in green letters.";
```

prints `The bottle is labelled "POISON" in green letters.` If you wish to output a literal tilde, you must use the construct @@126, (and, for completeness, a literal at-sign requires @@64).

In addition to embedding a quote within a string, the tilde is also employed to mean "not", in several different circumstances:

More information in the DM: §1.6 §1.8 §3.7 §39

"not equal" operator

For Example: `if (X ~= Y) { ... }`
Meaning: The condition is **true** if X is not equal to Y.

logical NOT operator

For Example: `if (~~(X == Y)) { ... }`
Meaning: The condition is **false** if X is equal to Y (this example is logically identical to that on the line above).

For Example: `X = ~~Y;`
Meaning: Invert Y's logical content. That is: if Y is **false** (0) then X becomes **true** (1); if Y is any other value then X becomes **false** (0).

bitwise NOT operator

For Example: `X = ~Y;`
Meaning: Invert Y's literal content, processing each bit individually. That is, if Y contains 41 ($0029 in hexadecimal) then X becomes -42 ($FFD6).

attribute resetting

For Example:

```
Object radio "radio"
           ...
   has  static ~on;
```

Meaning: The radio object is declared with its **on** attribute not set. Strictly unnecessary (since all attributes are unset by default), it's a handy way of reminding yourself that the attribute is likely to be set and reset during the game.

For example: `give radio ~on;`
Meaning: Clear (or unset, or remove) the radio's **on** attribute.

compiler control

For example: `-~S`
Meaning: Turn off the **S** switch, disabling Strict and Debug modes.

By the way, if you're uncertain of the difference between bitwise and logical operators, see this article by Sonja Kesserich.

How do I concatenate strings?

The short answer: you can't. There's no simple expression similar to the string manipulation commands provided by other languages. If you were thinking of using string concatenation, comparison, subsetting, case-folding or anything else on these lines, you need to rethink your approach. In practice, however, you'll almost certainly find that this isn't a problem: most Inform programmers don't even notice the lack of string handling capabilities.

Here's a longer answer (though it comes to the same thing). If you've come to Inform with programming experience in Basic, you might (mistakenly) expect to be able to do stuff like this:

```
Global X;
...
X = "Combining";
X = X + " some text";
print X;
```

Well, although Inform happily runs that code, the outcome is undoubtedly not what you were hoping for. What gets printed is a decimal number, guaranteed to be completely and utterly meaningless; here's why: X is an Inform variable, capable of holding any 16-bit number. The first line stores the string "Combining" at some suitable location within the Z-machine's memory, and puts the 16-bit address of that location into the variable X. The second line stores the string "some text" at another suitable location, and adds the 16-bit address of that location to the current contents of X (the address of the first location). The third line prints the new contents of X... which is garbage.

That is, adding together two strings makes about as much sense as saying "I've got a two-bedroom house at 54 High Street, and my fiance has a one-bedroom apartment at number 43. Therefore, when we get married, we'll own a three-bedroom residence at number 97." Life, and Inform, doesn't work like that.

If you think about it, you'll see why string comparison is also a non-starter:

```
X = "Combining";
if (X == "Combining") { ... }
```

That test is never ever going to be true (because it's comparing the address where the first string is stored with the address where the second string is stored). However, you *may* have more success with this approach:

```
Constant MYSTRING = "Combining";
...
X = MYSTRING;
if (X == MYSTRING) { ... }
```

Inform's string manipulation capabilities are virtually non-existent. A string -- one or more characters enclosed in quotes **"..."** -- is encoded by the compiler and packed into as few bytes as possible; these bytes are stored in a part of the Z-machine's memory which is dedicated to just this purpose, and referred to by the address of the first byte. Strings are like constants; they can be constructed only during compilation, and thereafter they don't vary. At run-time, the interpreter provides just two operations to handle strings: **print** unpacks the characters and displays them on the screen, and **print_to_array** unpacks the characters into the individual entries of a byte array. You'll notice that these are read-only operations; the Z-machine simply doesn't permit you to update a string after it's been created.

What's the difference between MyRoutine and MyRoutine()?

There are several reasons why you might write a standalone -- named -- routine, most commonly because you find yourself repeating the same chunk of processing in different places. For example, suppose that your game was time-oriented, sensitive to the difference between morning, afternoon and night; you could create a routine to facilitate this calculation:

```
[ PartOfDay i;
        switch (the_time) {       ! minutes since midnight
            360 to 719:  i = 1;   ! 06:00 - 11:59 (morning)
            720 to 1079: i = 2;   ! 12:00 - 17:59 (afternoon)
            default:     i = 0;   ! 18:00 - 05:59 (night)
            }
        return i; ];
```

Having written the routine once, you can now use it wherever you need to determine the difference:

```
if (PartOfDay() == 0) print "The moon"; else print "The sun";
print " shines thinly through the foliage overhead.";
```

When you call a routine, you supply any necessary arguments enclosed in parentheses after the routine's name. If the routine *doesn't* require an argument -- as is the case for PartOfDay -- it's important that you still supply a pair of empty parentheses. Consider these assignments:

```
X = PartOfDay();
Y = PartOfDay;
Z = Y()
```

More information in the DM: §1.7 §20

Afterwards, **X** contains the number (0, 1 or 2) returned by running the PartOfDay routine -- presumably the value that you were after -- whereas **Y** contains the routine's *address*; not nearly so useful. (Note, though, that **Z** is ok; like **X**, it contains 0, 1 or 2.)

What's the difference between a Directive and a Statement?

More information in the DM: §1.2 §2.1

Put simply, a directive controls what happens while the game is being compiled, whereas a statement controls what happens while it's being played. This distinction is perhaps most apparent with **Message** (a directive which outputs a message during compilation) and **print** (a statement which outputs a message at run-time). However, you don't often need to use **Message**; the directives that you'll most commonly encounter are **Include** -- for merging files during compilation -- and the set used for creating data structures: **Constant**, **Global** and **Array**; **Object** and **Class**; **Verb** and **Extend**. Oh, and **[...]**, the directive which wraps a succession of statements into a routine.

What's the difference between Include and #Include?

More information in the DM: §38

Not very much. They both merge in the contents of a specified file, with the distinction that **#Include** can be placed both within and outside the definition of a routine, whereas the plain **Include** can be placed only outside a routine. There's no down side, so you can happily use #Include in all cases. The same goes for the conditional compilation directives like **IfDef...IfNot...EndIf** -- you might as well use **#IfDef** and so on.

4 · Dabbling in Data

How do constants and variables differ?

A *variable* is a value which can change, possibly many times, while the game is being played; typical examples are **location** and **score**. A *constant* is a fixed value, set when when the game is compiled; it physically cannot change during play. For example, if you write:

```
  if (score == 100) { ... }
```

then pretty clearly the value of 'score' (a variable) might change at any time, but '100' (a constant) is never going to be anything other than the number between 99 and 101.

It's actually better programming practice to write this alternative form (which does, however, behave identically):

```
  Constant MAX_SCORE = 100;
  ...
  if (score == MAX_SCORE) { ... }
```

More information in the DM: §2.2 §2.3

There are two reasons for this. When you read through your game and happen upon that second **if** statement, the word **MAX_SCORE** makes it fairly apparent what the test is for. Also, if you later decided to change the game's maximum score, it's easier and more reliable to change the value assigned -- in one place -- to **MAX_SCORE** than it is to search though the program looking for values of '100' which *may* need to change. Here too you can see the advantage of naming constants in upper case -- it makes their distinction from variables visually apparent.

What can be stored in a variable?

Roughly speaking, the value that you can store in a variable is one of two things:

a **number** -- an integer in the range -32768..0..32767; for example:

```
        X = 42;
        X = MAX_CARRIED - 1;
```

the **other stuff** -- values that refer to an object, a string, a word in the dictionary, an action, a routine, and all the other internal Inform data types; for example (this list is not exhaustive, but does show most of the common examples):

```
        X = old_man;            ! a reference to an object
        X = "grumpy old man";   ! a reference to a string
        X = 'grumpy';           ! a reference to a dictionary word
        X = action_to_be;       ! a reference to a variable containing an action
        X = Initialise;         ! a reference to a routine (without running it)
        X = Initialise();       ! the value returned by running the routine
```

This difference is very important: arithmetic using numbers is fine, but arithmetic with Inform data types makes no logical sense. However, because Inform can't tell whether what's stored in a variable is a number or the other stuff, it doesn't stop you from performing meaningless operations, like dividing an object by a string. Be careful -- the onus is entirely on you to use your variables sensibly.

Bear in mind that the range of numbers that Inform handles is quite small. For example, if you multiply 1000 by 1000 in Inform, the result isn't one million: it's 16960, *and* you don't get any warning that an overflow has occurred.

More information in the DM: §1.4

What about fractions and decimals?

Inform deals only in integers -- whole numbers, without a decimal point. If you type this:

```
X = 25 / 4;
```

then Inform rounds down the answer and sets X to 6. Worse, if you type this:

```
X = 6.25;
```

then Inform looks at object number 6 (usually **compass**) and sets X to the value of its property number 25 (usually **list_together**). Basically, the value "six and a quarter" doesn't exist in Inform.

If you really need them, some mathematical extensions are available. Almost always, however, you'll find that you can rethink what you're trying to do so that integer arithmetic is sufficient.

How do global and local variables differ?

You can declare a *global variable*, capable of holding a single value, with this directive:

```
Global globalCount = 42;
```

More information in the DM: §1.5 §1.7

A global variable is accessible from anywhere in the program, which is what primarily distinguishes it from a *local variable* -- one declared within a routine -- which can be used only within that routine:

```
[ myRoutine singleCount; ... ];
```

The other significant point of difference is that, once set, the value stored in a global variable remains the same until you explicitly change it. Local variables, on the other hand, are reset every time that you call the routine, either by an argument supplied as part of the call or to zero. Think of it this way: any routine can define up to seven local variables, which double as the incoming argument values:

```
[ MyRoutine V1 V2 V3 V4 V5 V6 V7; ... ];
```

Each time that you call myRoutine() -- without any arguments -- all seven variables are initialized to zero before the routine's code is executed. If instead you call, for example, myRoutine(100,200,300) then V1, V2 and V3 are initialized to those three values, while the remaining four variables are set to zero.

What does an array provide?

An array behaves like a global variable, but is capable of holding several values rather than just one. So, you could write, for example:

```
Array multiCount --> 10;
```

More information in the DM: §2.4

to give yourself ten global variables multiCount-->0, multiCount-->1 ... multiCount-->9. Now clearly there's no advantage whatsoever in bundling together a pile of unrelated globals into an array -- you're much better off defining separate globals with meaningful names. The power of an array comes when you find yourself in a position to manipulate the appropriate variable using that 'index' number at the end.

It's rare, however, for newcomers to find themselves in that position. You can safely ignore the use of arrays until you've written a fair bit of Inform code and have built up your confidence in the way that the language works.

There's helpful introduction by Zak McKracken and Sonja Kesserich to creating and initializing arrays

Note that Inform has no concept of a local array, one that is accessible only within a routine.

What's an 'unsigned' number?

In an Inform game, each number is stored as a pattern of binary digits (bits), with sixteen bits being grouped together to form a 'word'. Since each bit in a word can have only two states -- 0 and 1 -- the sixteen of them can be combined in 65536 (=2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2) different patterns. By a widely-accepted convention, those patterns are logically associated with two series of 65536 decimal numbers: *signed* numbers ranging from -32678 upwards through 0 and on to 32767; and *unsigned* numbers ranging from 0 up to 65535. Note that the words 'signed' and 'unsigned' refer *only* to the way that a bit pattern is interpreted as a decimal number, not to the way that it is stored.

Here are some of the patterns, showing how they are associated with signed and unsigned numbers (**$$** is Inform's way of introducing a binary value). You'll notice that in the first half of the table -- when the leftmost bit in the pattern of sixteen is 0 -- the signed and unsigned values are the same. In the second half -- when the leftmost bit is 1 -- they differ radically. You won't be surprised to hear that that leftmost bit is commonly known as the 'sign bit'.

Binary bit pattern	Signed	Unsigned
	decimal	decimal
$$0000000000000000	0	0
$$0000000000000001	1	1
$$0000000000000010	2	2
$$0000000000000011	3	3
$$0000000000000100	4	4
...	...	...
$$0111111111111100	32764	32764
$$0111111111111101	32765	32765
$$0111111111111110	32766	32766
$$0111111111111111	32767	32767
$$1000000000000000	-32768	32768
$$1000000000000001	-32767	32769
$$1000000000000010	-32766	32770
$$1000000000000011	-32765	32771
$$1000000000000100	-32764	32772
...	...	...
$$1111111111111100	-4	65532
$$1111111111111101	-3	65533
$$1111111111111110	-2	65534
$$1111111111111111	-1	65535

More information in the DM: §1.4

Actually, you very rarely need worry about this; Inform treats almost all numbers as *signed*, so if you take any two values lying in the range -32768..0..32767 and add them, subtract them, multiply them or whatever, you'll automatically get the expected answer (providing, of course, that the answer is also in that range).

There's one -- rare -- situation where the difference between signed and unsigned numbers *is* important: the conditional operators < and > perform a *signed* comparison. To explain what this means, imagine that the variable V1 contains $$0000000000000010, the variable V2 contains $$1111111111111100, and that you're testing if (V1 > V2). Now, since > performs a signed compare, V1's value is taken as 2, and V2's value is taken as -4; the test becomes (2 > -4), which correctly evaluates to **true**. However... if for some obscure reason the values in V1 and V2 had been object addresses, or pointers, or some other unsigned numbers, then the test *should have been* treated as (2 > 65532), which of course ought to evaluate to **false**, so you'll get the wrong answer. The moral is: if you need to test unsigned numbers, use **UnsignedCompare()** rather than < and >.

What exactly are 'true' and 'false'?

There are two answers to this question. Physically, **true** is a constant with a value of '1', and **false** is a constant with a value of '0'. Logically, things are slightly different: 'logical' false is still zero, but 'logical' true is *any* non-zero value, not just 1.

More information in the DM: §1.4 §1.8

Testing whether myVar is 'true'	**Testing whether myVar is 'false'**
if (myVar) { … }	**if (~~myVar) { … }**
if (myVar == true) { … }	**if (myVar == false) { … }**
if (myVar ~= false) { … }	**if (myVar ~= true) { … }**
if (~~myVar == false) { … }	**if (~~myVar == true) { … }**
if (~~myVar ~= true) { … }	if (~~myVar ~= false) { … }
if (~~(myVar == false)) { … }	**if (~~(myVar == true)) { … }**
if (~~(myVar ~= true)) { … }	if (~~(myVar ~= false)) { … }
if ((~~myVar) == false) { … }	if ((~~myVar) == true) { … }
if ((~~myVar) ~= true) { … }	if ((~~myVar) ~= false) { … }

This distinction matters when you're using conditional statements like **if**. Suppose that you wish to test a true/false variable myVar; there are about eighteen ways in which you could construct the **if** statement. The nine variations on the left are all triggered when myVar contains **true** (1), and the matching nine on the right are all triggered when myVar contains **false** (0).

However, suppose that myVar happens to contain 2, or 100, or -1; all values which, being non-zero, are logically true. Six of those eighteen statements don't work as you might have hoped: the three **red** statements on the left will fail to trigger, and the three on the right will be triggered unexpectedly. If we also discount the other eight shaded statements as being over-complex, that leaves only four 'reliable' ways of constructing the statement.

The bottom line is: test against **false** rather than against **true**. If you want to do something when a value is logically true, use either of:

```
if (myVar) { ... }
if (myVar ~= false) { ... }
```

and the complementary tests, when a value is logically false, should use either of:

```
if (~~myVar) { ... }
if (myVar == false) { ... }
```

It's occasionally worth remembering that *all* Inform conditions (for example, `a >= b` and `MyObj in location`) evaluate to either 1 or 0, values which you can use in an expression. For example, here's a simple routine, similar to **UnsignedCompare()**, to compare two numbers:

More information in the DM: Table 1B

```
[ Compare a b;
    if (a > b) return 1;
    if (a < b) return -1;
    return 0;
];
```

and here's the same thing using a little arithmetic cunning:

```
[ Compare a b;
    return (a > b) - (a < b);
];
```

How do I return data values from a routine?

More information in the DM: §1.7 §3.5

Inform has two types of routine: a **standalone** routine which is specified independently, and an **embedded** routine which is specified as a property value within an object definition. Every time that you call a routine, of either type, you get back a single **return value** (which you're free to ignore if you want to).

This embedded routine	Can be called like this	And returns the value

`Object myObject`
`  with ...`
`       myProperty [;`
`          print "Hello, world!";`
`       ],`
`       ... ;`
	`myObject.myProperty();`
	`x = myObject.myProperty();`
	`if (myObject.myProperty())`
	`   { ... }`
	`etc`
		false. Note the difference between the value returned at the end of a standalone routine, and that returned at the end of an embedded routine like this.

`Object myObject`
`  with ...`
`       myProperty [;`
`         print "Hello, world!";`
`         rtrue;`
`       ],`
`       ... ;`
		true.

`Object myObject`
`  with ...`
`       myProperty [;`
`         print "Hello, world!";`
`         rfalse;`
`       ],`
`       ... ;`
		false.

`Object myObject`
`  with ...`
`       myProperty [;`
`         print "Hello, world!";`
`         return;`
`       ],`
`       ... ;`
		true.

`Object myObject`
`  with ...`
`       myProperty [;`
`         print "Hello, world!";`
`         return` *`expression`*`;`
`       ],`
`       ... ;`
		given by evaluating the *expression*.

This standalone routine	Can be called like this	And returns the value

`[ myRoutine;`
`print "Hello, world!";`
`];`
	`myRoutine();`
	`x = myRoutine();`
	`if (myRoutine()) { ... }`
	`etc`
		true. Note the difference between the value returned at the end of a standalone routine like this, and that returned at the end of an embedded routine.

`[ myRoutine;`
`print "Hello, world!";`
`rtrue;`
`];`
		true.

`[ myRoutine;`
`print "Hello, world!";`
`rfalse;`
`];`
		false.

`[ myRoutine;`
`print "Hello, world!";`
`return;`
`];`
		true.

`[ myRoutine;`
`print "Hello, world!";`
`return` *`expression`*`;`
`];`
		given by evaluating the *expression*.

As mentioned above, you can pass up to seven arguments to a routine. Here's a simple routine, and an example of it being called:

```
 [ myRoutine V1 V2;
     V1 = V1 + V2;
     print V1;
 ];

 ...

myRoutine(100,200);
```

The routine adds together the values of its two arguments, stores the result in the first argument, and then prints the result -- 300 in our example. Now consider this second call:

```
num1 = 100; num2 = 200;
myRoutine(num1,num2);
```

How does this second example differ from the first? It doesn't: the two behave identically. The important thing to note is that, after the call, **num1** still contains 100. Although the value of local variable **V1** is changed from 100 to 300 within the routine, this change *is not* reflected back to the variable **num1** (in technical terms, Inform uses 'call by value' rather than 'call by reference'). In summary:

- You can supply up to seven argument values when you call a routine. You can change those values within the routine, but your changes are lost when you return; they're not visible outside of the routine.
- You get back a single value from a routine call. This value can be set *explicitly* by a **return**, **rtrue** or **rfalse** statement, or *implicitly* by the '**]**' which terminates the routine. The implicit return value is **true** from a standalone routine, and **false** from an embedded routine.

Suppose that you need to return more than one value from a routine; are you stuck? As it happens, no you're not; there's a technique which enables you to return multiple values, but it requires you to learn about arrays, so you might want to leave it until you're fairly comfortable with Inform. The trick is to pass the name of an array as an argument; your routine then has full read/write access to all of the entries of the array, so you can set return values into as many of those entries as you need. An example will help explain what this means.

Suppose that you need a routine which takes a single numeric argument 'msm' -- the time as the number of minutes since midnight -- and returns three values: the hour, the minute of the hour, and an am/pm indicator. First, we'll write the routine:

```
  [ getClockTime msm result;                ! 'result' argument is an array
      if (msm < 0 || msm > 1439) rfalse;      ! error: 'msm' argument out of
range
      result-->0 = msm / 60 % 12;           ! 'hours' value   0..11
      if (result-->0 == 0) result-->0 = 12;   ! 'hours' value   12, 1..11
      result-->1 = msm % 60;                ! 'minutes' value 0..59
      result-->2 = (msm > 719);             ! false for 'am', true for 'pm'
  ];
```

then we'll define an array with three entries, and show the routine in use:

```
  Array clockTime --> 3;

  if (getClockTime(the_time, clockTime)) {
      if (clockTime-->0 < 10) print "0";      ! leading zero for 'hours'
      print clockTime-->0, ":";
      if (clockTime-->1 < 10) print "0";      ! leading zero for 'minutes'
      print clockTime-->1;
      if (clockTime-->2) print " pm"; else print " am";
  }
```

Using this technique (and a big enough array), you can overcome both the limit on seven arguments to a routine, and the restriction of only a single return value.

Where do 'random' numbers come from?

First, what do we mean by a 'random' number? In Inform terms, it's a positive integer in the range 1..32767, ideally chosen completely by chance. Imagine an enormous roulette wheel, with 32767 numbered pockets rather than the usual 37 or 38. Spin the wheel and see where the ball lands: that's our random number.

Except that, using software, it turns out to be really hard to replicate the true randomness of a perfectly-balanced roulette wheel, where at each spin there's exactly one chance in 32767 of a particular number turning up. Instead, we use a pseudo-random number generator (PRNG), an algorithm that generates a sequence of numbers whose elements are *approximately* independent of each other. Most simple PRNGs (that is, whose size and complexity is appropriate for a text adventure game) generate each random number by applying some formula to the number that was generated last time. Let's illustrate this with a trivial example; to make things easier to understand, we'll restrict our random numbers to the range 1..10:

```
  Global prev_rand = 1;

  [ trivial_PRNG;
      prev_rand = prev_rand + 7;
      if (prev_rand > 10) prev_rand = prev_rand - 10;
      return prev_rand;
  ];
```

Calling **trivial_PRNG()** repeatedly returns a sequence of numbers -- 8 5 2 9 6 3 10 7 4 1 8 5 2 9 6 3 and so on -- which certainly looks a bit random. Closer inspection reveals a couple of problems: the numbers are alternately even and odd, and the whole sequence starts again every time we reach the number 8. The first problem can be solved by using a more sophisticated formula than our trivial "just add 7", but the second one is an intrinsic limitation of this sort of PRNG (in technical terms, thanks to wikipedia, "Because any PRNG run on a deterministic computer is a deterministic algorithm, its output will inevitably have one property that a true random sequence would not exhibit: guaranteed periodicity."). In practice, the fact that any sequence of pseudo-random numbers is certain, sooner or later, to repeat itself isn't usually a problem; what's much more important is using a PRNG algorithm, better than our trivial_PRNG(), which ensures that there's no obvious pattern (such as even/odd alternation, or lots of low numbers followed by lots of high numbers) to the sequence of generated numbers.

Because a PRNG produces each new number by applying some algorithm to the number that it returned on the previous occasion, it's necessary to establish a starting value before calling the PRNG for the first time: this initial number is called a **seed**, and we sometimes use the phrase "seeding the generator" to refer to the PRNG initialisation process. Just storing a constant value, as we did with `prev_rand = 1;`, isn't normally a sensible idea, because that would mean our sequence of random numbers always starting from that same constant value and continuing in a predictable fashion. Instead, programs attempt to seed the generator by picking an initial seed value, for example the current time in milliseconds, which should be different for each run.

So, based on our rough-and-ready simplification of what 'randomness' means, we can see how it applies within Inform. The function **random()** -- it's built into the Inform language rather than being defined in the library -- returns a random value, and may be called in two formats:

More information in the DM:§1.14

```
  random(6);                   ! Format 1

  random(10, 20, 30, 40);     ! Format 2
```

The first format returns 1, 2, 3, 4, 5 or 6: that is, a random positive integer in the range 1 to the specified upper limit. Other examples: `random(1)` always returns 1, `random(2)` returns either 1 or 2, and `random(32767)` returns a value in the range 1..32767 (the largest positive integer).

The second format returns 10, 20, 30 or 40: that is, a random selection from the set of specified values; in this instance, we could equally have used `10*random(4)`. Other examples:

`random(1,1,1,2)` returns either 1 or 2 (with a distinct bias towards 1),

`random(2,3,5,7,11,13,17,19,23,29,31,37,41,43,47)` returns a small prime number,

`random(n_obj,s_obj,e_obj,w_obj)` returns a compass object,

`random("circle","triangle","square","diamond","pentagon")` returns the (packed) address of a string, and

`random('red','green','blue')` returns the (unpacked) address of a dictionary word.

Note, by the way, that this format accepts only constant arguments – for instance, `random(myVar1,myVar2,myVar3)` isn't allowed -- and additionally is not subject to the normal Inform restriction that a routine is limited to seven arguments. Both of these oddities are side-effects of the compiler transforming what looks like a normal routine call into some customised Z-code construct.

As stated above, the PRNG is automatically seeded at the start of each game to what is hopefully an unpredictable value, thus ensuring that a game which includes calls to **random()** will behave differently each time that it's run. Usually that's what the author wants: there's little point in writing a puzzle involving chance events if their outcome is always identical and so can be anticipated by the player. There *is* an exception to this preference for unpredictability, however, and that's when you're testing a game prior to release. In this situation, it's often useful to REPLAY a script containing a master-list of commands which progress through the entire game, exercising every feature. For the script to be unfazed by 'random' events, you need those events to happen identically on each replay: that is, you need your sequence of random numbers to become predictable. This is easily achieved by seeding the PRNG to a constant value rather than one which changes each time. Inform provides three more **random()** formats to cover this situation:

More information in the DM: §7.1

```
  random(-1000);               ! Format 3 (argument's absolute value in range
1000..32767)

  random(-100);                ! Format 4 (argument's absolute value in range
1..999)

  random(0);                ! Format 5
```

The third format, which returns zero, seeds the PRNG with a starting value of 1000. Subsequent calls using Formats 1 and 2 then return a predictable sequence of 'random' numbers which are repeatable every time that the game runs.

More information in the Z-Machine Standards Document: §2.4 §15

The fourth format, which also returns zero, seeds the PRNG in such a way that subsequent calls using Formats 1 and 2 then return values based upon the simple rising sequence 1,2,...,99,100,1,2,...,99,100,1.2...; this too is repeatable every time that the game runs.
The fifth format, which again returns zero, re-seeds the PRNG so that subsequent calls using Formats 1 and 2 revert to unpredictable random results.

At this point, we need to highlight a couple of potential availability problems. Format 4, whose behaviour is undocumented in the DM4, is Graham's 'suggestion' hidden in the Remarks at the end of Section 2 of the Z-Machine Standards Document; this algorithm is not honoured by all interpreters, which tend to treat this format as though it was Format 3. Format 5, similarly undocumented in the DM4, is flagged in Section 15 of the Z-Machine Standards Document as being considered illegal by most interpreters, though this is much less true today (the Standard dates from 1997). The important point is: just because the PRNG in the interpreter that *you* use for testing your game behaves in a certain fashion, you cannot assume that your audience will encounter the same behaviour in the interpreters that *they* use for playing your game.

And the difference in behavior associated with Format 4 leads to another problem. The library defines the debugging verb RANDOM thus:

```
  [ PredictableSub i;
      i = random(-100);
      "[Random number generator now predictable.]";
  ];

  Verb meta 'random'
      *                            -> Predictable;
```

If you test your game using an interpreter like Frotz, which honours Format 4 by generating a simple rising sequence, then you'll get very different behaviour than if you test with an interpreter which doesn't distinguish between Formats 3 and 4, generating a true (albeit predictable) random sequence. The library doesn't provide a standard way of specifying this latter sequence, an omission which you might wish to remedy by added this at the end of your game:

```
  #Ifdef DEBUG;

  [ RandomUnpredictableSub;
      random(0);
      "[Random number generator now unpredictable.]";
  ];

  [ RandomSerialSub s;
      if (s == 0) s = 100;
      random(-s);
      "[Random number generator now serial.]";
  ];

  [ RandomRepeatableSub s;
      if (s == 0) s = 1000;
      random(-s);
      "[Random number generator now repeatable.]";
  ];

  [ RandomSeedSub;
      if (noun < 0) noun = -noun;
      switch (noun) {
        0:          RandomUnpredictableSub();
        1 to 999:   RandomSerialSub(noun);
        default:    RandomRepeatableSub(noun);
      }
  ];

  Extend 'random' replace
      *                           -> RandomSerial
      * 'serial'/'s//'            -> RandomSerial
      * 'repeatable'/'r//'        -> RandomRepeatable
      * 'unpredictable'/'u//'     -> RandomUnpredictable
      * number                    -> RandomSeed;

  #Endif; ! DEBUG
```

Finally, here are a couple of ways of asserting some control over your random numbers despite the vagaries of interpreters' PRNGs. Jim Fisher has an article on Randomizing Random, which shows a technique for obtaining a good random seed value. Roger Firth's **random.h** extension, available from the Archive, provides a complete replacement for the PRNG, one that behaves the same across all interpreters.

5 · Operating on Objects

Is a 'room' a special sort of object?

No: a 'room' is simply an Inform object which the player can enter by some means, and which doesn't have a parent. There’s nothing more to it than that.

How does Inform distinguish nouns from adjectives?

It doesn't. In an object's **name** property, you need to list *all* of the words that a player might think to use when referring to that object. For example, in this object, introduced when talking about apostrophes and quotes:

```
Object  old_man "grumpy old man"
  with  name 'man' 'grumpy' 'cross' 'old' 'elderly' 'ancient' 'gnome',
        description "The grumpy old man stands silently in the corner.",
        ...
  has   animate;
```

More information in the DM: §28

you'll see there are two nouns -- 'man' and 'gnome' -- and five adjectives. The player could use these seven words (also THE which is always available) in every possible combination, both sensible (GRUMPY OLD MAN, THE GNOME, THE ELDERLY MAN, CROSS GNOME) and not so sensible (THE ANCIENT CROSS, GRUMPY GRUMPY, MAN OLD ELDERLY ANCIENT GNOME). In all of these cases, Inform will infer that the player is referring to the "grumpy old man" object. As it happens, players have become accustomed to this behaviour, and don't find it much of a problem.

Occasionally, you'll create a set of objects with similar names, in which circumstance having Inform react to nouns more strongly than to adjectives is helpful to players. (Without such special treatment, your players may have difficulty picking up, for example, a JEWEL which occurs in the same location as a JEWEL BOX.) Neil Cerutti's `pname.h` from the Archive offers a good solution in these circumstances.

When should I use scenery/static/concealed attributes?

You affect an object's behaviour using these attributes. The main differences are:

attribute	Does LOOK m ention it?	Can you TAKE it?	Does TAKE ALL attempt it?
scenery	no	no -- "hardly portable"	yes (annoyingly)
static	yes	no -- "fixed in place"	yes
concealed	no	yes -- listed in your inventory, but disappears again if you drop it	no
none of these	yes	yes	yes

One of the main uses of **concealed** is for something that can be 'discovered' during the game, for example when you SEARCH another object. Using this technique, you might use an **after** property on the object being searched to remove the hidden object's **concealed** attribute, thereby bringing it into view. The alternative method is to use that same **after** property to bring the hidden object into the room from elsewhere; this has the disadvantage that second-time-around players still need to SEARCH even though they know that the hidden object is present. (On the other hand, if you allow them this shortcut, you need to remove the **concealed** attribute anyhow if they do TAKE without SEARCH.)

On the subject of **scenery** and **static**, you occasionally run into trouble with the parser being too helpful. If the player types a command like TAKE without specifying an object, the parser usually asks "What do you want to take?", which is fine. If, however, there is only one object in scope, the parser guesses that that must be what you want, even if it's non-takeable. For example:

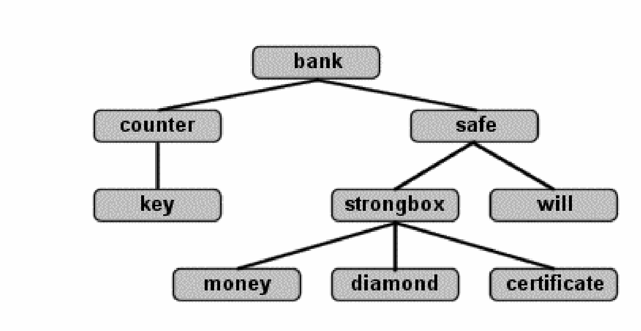
```
>TAKE
(the writing desk)
The writing desk is too heavy for that.
```

The best way to avoid this is by ensuring that there's always a choice of objects with the same attribute level -- two without **static**/**scenery**, or failing that two with **static**, or as a last resort two with **scenery**.

How can I tell that one object is 'within' another?

A typical Inform program defines dozens of objects: rooms, furniture, pieces of scenery, Non-Player Characters (NPCs), artefacts of all shapes and sizes. Rather than floating independently in some abstract space, most of those objects are linked by formal (but fluid) relationships. An important part of Inform programming is managing the changes in those relationships: keeping track of which objects are 'in' or 'on' other objects at each turn in the game.

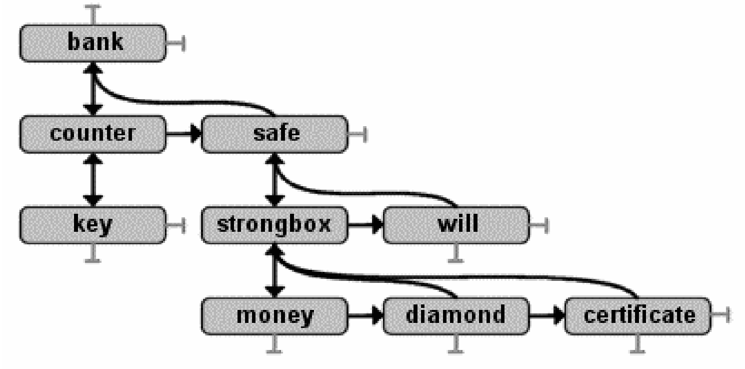
For example, you might visualize the relationship between several objects in a bank thus. In this diagram, the money, diamond and certificate are in the strongbox, which is in the safe (along with the will), and the safe is in the same room as the counter, on which is a key. It's common to talk about an object's parent and its children: thus, the safe's parent is the bank, and its children are the strongbox and the will. If the player was to enter the bank and type TAKE THE KEY, the diagram would need redrawing to show the player as a child of the bank, and the key as a child of the player.



The diagram above shows a conceptual representation in which each object has a single parent object (or **nothing**) above it, and zero, one or more child objects below it. However, this representation doesn't precisely reflect the way in which Inform stores its data. The relationship between Inform objects -- the object tree -- is actually defined by three pointers. Each object has an 'upwards' pointer towards its parent, a 'downwards' pointer towards its child, and a 'sideways' pointer towards its sibling; a pointer contains the constant **nothing** if there is no object above, below or to the side.

This system of pointers means that the set of objects in the bank is actually connected as shown alongside. You might notice that there is a small discrepancy between the conceptual model -- in which an object like the strongbox can have several children -- and the physical model, where an object has one (eldest) child, with any younger children being linked out to the side. The built-in functions **parent()**, **sibling()** and **child()** reflect this physical view:

```
parent(bank) == nothing
sibling(bank) == nothing
child(bank) == counter

parent(counter) == bank
sibling(counter) == safe
child(counter) == key

parent(safe) == bank
sibling(safe) == nothing
child(safe) == strongbox
```



and so on.

You can change object relationships with the **move** and **remove** statements, but it's relatively rare that you need to do so -- most of the time, the Library does everything necessary to move objects around the tree. What you frequently *do* need is the ability to discover where an object is currently located. You can test whether one object is a child of another object -- its parent -- using either of:

```
if (parent(c_obj) == p_obj) { ... }
if (c_obj in p_obj) { ... }
```

More information in the DM: §3.2 §3.4

You *can't* reliably use this test:

```
if (c_obj == child(p_obj)) { ... }
```

because the **child()** function returns only an object's *eldest* child, rather than a list of *all* children; for example (safe == child(bank)) evaluates to **false**.

The first two tests, although reliable, are also limited in scope: they concern themselves only with an object's *immediate* children. So, (parent(safe) == bank) and (safe in bank) both evaluate to **true**, but (parent(strongbox) == bank) and (strongbox in bank) both evaluate to **false** -- the strongbox object is a grandchild of the bank object, not a direct child.

To get round this problem, you can use the Library routine **IndirectlyContains()**:

```
if (IndirectlyContains(p_obj, c_obj)) { ... }
```

This routine returns **true** if *p_obj* is a parent, or grandparent, or great-grandparent, or great-great-grandparent..., of *c_obj*; that is, if *c_obj* lies *anywhere* below *p_obj* in the tree. So, IndirectlyContains(bank,strongbox) evaluates to **true**.

A related routine, **CommonAncestor**(*c_obj1*, *c_obj2*), returns the nearest object which has a parental relationship to both *c_obj1* and *c_obj2*, or **nothing**. For example, CommonAncestor(diamond,will) would return the safe.

How do I get rid of an object in mid-game?

You don't actually *delete* an unwanted object during the course of a game; you just reposition it out of the way so that the player won't ever encounter it. You usually do this by making its parent **nothing**, which detaches it from the object tree. However, you can't type `move myObj to nothing;` since **nothing** isn't an object; instead you use:

More information in the DM: §3.4

```
remove myObj;
```

The removed object now lurks out of sight, until either the game ends or you choose to resurrect it. To bring the object back into play, you simply need something like:

```
move myObj to location;
```

That's really all there is to it, though while we're here we'll mention a few related points. You can test whether an object has been **remove**d by either of these:

```
if (myObj in nothing) { ... }
if (parent(myObj) == nothing) { ... }
```

but what you *can't* so easily do is identify all of the removed objects. This is a valid statement:

```
objectloop (x && x in nothing) { ... }
```

but it finds too much: several Library objects and all of the rooms, as well as any objects which you've shifted out of play. If you need to loop through out-of-play objects, a better approach is *not* to use the **remove** statement. Instead, create an inaccessible room -- commonly called "Limbo" -- and **move** unwanted objects into it. This is all you need:

```
Object  limbo "Limbo"
  with  description "How the hell did you get in here?";
  ...
move myObj to limbo;
  ...
objectloop (x in limbo) { ... }
```

If you've given myObj a **found_in** property -- so that it magically pops up in various rooms as the player visits them -- you've got a little more work to do. If you just **move** or **remove** such an object out of play, the Library will promptly transport it back again. You need to prevent this, by applying the **absent** attribute:

More information in the DM: §8

```
move myObj to limbo;    ! or: remove myObj;
give MyObj absent;
```

Finally, the rather more complex business of creating and deleting objects during play. We'll skip most of the detail and just sketch the outline of dynamic object manipulation. First, you need a class from which to create your objects (the "10" specifies the maximum number which can be in play concurrently):

More information in the DM: §3.11

```
Class   myObj(10)
  with  ... ;
```

and then you use statements like these to add a new object to the game, and later to get rid of it again:

```
x = myObj.create();
...
myObj.destroy(x);
```

Note that **destroy()** works only for objects which have previously been added using **create()**, not for normal objects which have existed throughout the game.

What are the various 'description' properties for?

Two of the more popular commands during a game are LOOK (or L) and EXAMINE or (X). Here's a typical example:

```
>LOOK

Dingy hall
Steps lead down into darkness.

You can see a trunk (which is closed) and a flashlight here.

>EXAMINE THE TRUNK
The trunk is large enough to get inside.
```

Here you can see the ubiquitous 'description' property doing its stuff: LOOK displays the current location's name, its **description** and a list of its notable contents, while EXAMINE displays the **description** of the specified object:

```
Object  hallway "Dingy hall"
  with  description "Steps lead down into darkness.",
        d_to cellar,
  has   light;

Object  -> "trunk"
  with  name 'trunk' 'chest' 'box',
        description "The trunk is large enough to get inside.",
  has   static container enterable openable ~open;
```

```
Object  -> "flashlight"
   with  name 'flashlight' 'torch',
         description "It's a battery-powered flashlight, of the sort that
                      switches on and off.",
         after [;
           SwitchOn:  give self light;
           SwitchOff: give self ~light;
         ],
   has   switchable ~on ~light;
```

The library provides a number of standard object properties which are used in specific circumstances to display descriptive information about the object; **description** is the one most commonly found, but the others all have their place.

More information in the DM: §26

This table summarises their usage:

Property	For a room	For an object
description	The first text (after the room's name) displayed by LOOK in that room, and on initial entry to the room.	Displayed by EXAMINE of the object. This is the only property which is consulted by the EXAMINE command; all of the other entries in this table are associated with the LOOK command.
inside_description	Displayed after the room's **description** text.	For an **enterable** object which the player is inside, displayed after the parent room's **description** text (if the room is still visible) or instead of it (if not). Over-rides **initial** and **when_open**/**when_closed**.
describe	The first text (after the room's name) displayed by LOOK in that room, and on initial entry to the room. Over-rides **description** (but provides no additional functionality and so is effectively redundant for a room).	Displayed after the parent room's **description** text. If this property is a string or a routine returning true, any of the properties below are ignored.
initial	Displayed *before* the room's name when the player moves to that room.	For an object which hasn't **moved**, displayed after the parent room's **description** text. Once the object has **moved**, this property is ignored and the object is included in the "You can [also] see..." list.
when_on	Ignored.	For a **switchable** object which has **on**, displayed after the parent room's **description** text. Over-rides **initial**.
when_off	Ignored.	For a **switchable** object which hasn't **on**, displayed after the parent room's **description** text. Over-rides **initial**.
when_open	Ignored.	For a **door** or **container** object which has **open**, displayed after the parent room's **description** text. Over-rides **initial** and **when_on**/**when_off**.
when_closed	Ignored.	For a **door** or **container** object which hasn't **open**, displayed after the parent room's **description** text. Over-rides **initial** and **when_on**/**when_off**.

none of these	-	Displayed in the list of "You can [also] see..." objects at the end of the room description (except for **concealed** or **scenery** objects).

It's hard to predict the precise presentation -- trial and error is your best bet -- but as a rule-of-thumb, the library automatically prints a newline *before* each property value (except for **describe**), and also prints one *after* each property whose value is a string (if a property value is a routine, the terminating newline is under your control).

Here's our previous example, now ornamented with a few of these properties:

```
Object  hallway "Dingy hall"
  with  description "Steps lead down into darkness.",
        d_to cellar,
  has   light;

Object  -> "trunk"
  with  name 'trunk' 'chest' 'box',
        description "The trunk is large enough to get inside.",
        inside_description "From the inside, the trunk doesn't
                           now seem so big.",
        when_open "An open trunk sits in one corner.",
        when_closed "The trunk in the corner looks like it can be opened.",
  has   static container enterable openable ~open;

Object  -> "flashlight"
  with  name 'flashlight' 'torch',
        description "It's a battery-powered flashlight, of the sort
                    that you switch on and off.",
        after [;
          SwitchOn:  give self light;
          SwitchOff: give self ~light;
        ],
        when_on "The flashlight provides a faint glimmer.",
  has   switchable ~on ~light;
```

And this is the result:

```
>LOOK

Dingy hall
Steps lead down into darkness.

The trunk in the corner looks like it can be opened.

You can also see a flashlight here.

>SWITCH ON THE FLASHLIGHT
You switch the flashlight on.

>OPEN THE TRUNK
You open the trunk.

>LOOK

Dingy hall
Steps lead down into darkness.

An open trunk sits in one corner.

The flashlight provides a faint glimmer.

>ENTER TRUNK
You get into the trunk.

>LOOK

Dingy hall (in the trunk)
Steps lead down into darkness.

The flashlight provides a faint glimmer.
```

```
From the inside, the trunk doesn't now seem so big.
```

How are plural objects managed?

While most of your game objects will be singular -- A SHINY SWORD, AN ANCIENT BRASS LANTERN, THE GRUMPY OLD MAN -- you will occasionally need to represent plurals: multiple objects such as A HANDFUL OF BEANS, THREE GOLD COINS, A BRACE OF PHEASANTS. The Library provides several standard properties and attributes which are useful when dealing with plural objects; here are some of the basic techniques:

Mass noun (singular)

For example: gunpowder, machinery, foliage

Referenced by the player as: THE GUNPOWDER, SOME GUNPOWDER, IT

Referenced by the game as:

```
You can see some gunpowder here.
You see nothing special about the gunpowder.
That's not something you can close.
You can't wear that!
```

Example:

```
Object  -> "gunpowder"
  with  name 'gunpowder' 'blasting' 'powder',
        article "some";
```

More information in the DM: §26

Use the **article** property: the game understands the object as singular, but incorporates the possibility of using a special prefix in place of "a" or "an" -- in this case, "some". The **article** property lets you define whatever word or group of words will best suit your needs. In the example, you could replace "some" with "lots of", "several tons of", "a few grains of", etc.

Mass noun (plural)

For example: instructions, grapes, scissors

Referenced by the player as: THE INSTRUCTIONS, SOME INSTRUCTIONS, THEM

Referenced by the game as:

```
You can see some instructions here.
You see nothing special about the instructions.
They're not something you can close.
You can't wear those!
```

Example:

```
Object  -> "instructions"
  with  name 'instructions',
  has   pluralname;
```

More information in the DM: §26 §29

Use the **pluralname** attribute: the game marks the object as plural, automatically changing pronouns and articles as needed.

The **name** property of the object should list only plural names and synonyms, because the **pluralname** attribute tells the parser and the library to consider the object as plural in all cases. Consider the following (erroneous) code:

```
  Object  -> "instructions"
    with  name 'sheet' 'of' 'instructions',
          article "a sheet of",
    has   pluralname;
```

Here the player could type, for example, OPEN SHEET (which is singular) and get the mismatched plural response "They're not something you can open.". For a way round this, see the next example.

Singular/plural noun

For example: string of pearls, sheet of instructions, bunch of grapes

Referenced by the player as: THE STRING, THE STRING OF PEARLS, THE PEARLS

Referenced by the game as: `You can see {a string of pearls|some pearls on a string} here.` `{That's|They're} not something you can close.`

Example:

More information in the DM: §26 §28

```
 Object ->
   with name 'string' 'of' 'pearls' 'pearl',
        parse_name [ wd num singular;
          for (wd=NextWord() : WordInProperty(wd,self,name) : wd=NextWord()) {
              num++;
              if (wd == 'string') singular = true;
              }
          if (num) {
              if (singular) give self ~pluralname;
              else          give self pluralname;
              }
          return num;
          ],
        short_name [;
          if (self has pluralname)
               print "pearls on a string";
          else print "string of pearls";
          rtrue;
          ],
     has ~pluralname;
```

If you supply a **parse_name** property, it overrides any **name** property. These three examples work identically:

```
      name 'string' 'of' 'pearls',

      parse_name [ wd num;
                   for (wd=NextWord() :
                       wd=='string' or 'of' or 'pearls' :
                           wd=NextWord()) num++;
                   return num; ],

      name 'string' 'of' 'pearls',
      parse_name [ wd num;
                   for (wd=NextWord() :
                       WordInProperty(wd,self,name) :
                           wd=NextWord()) num++;
                   return num; ],
```

The advantage of a **parse_name** property is the capability of recognising certain trigger words, and then modifying or extending the standard behaviour. Here, EXAMINE STRING makes the object singular, while EXAMINE PEARLS makes it plural.

The printable name is supplied by a **short_name** property rather than being given on the object's header line, so that it too can toggle between singular and plural forms.

Enumerated set of indistinguishable nouns #1

For example: sticks of dynamite, throwing knives

Referenced by the player as: THE STICK, TWO [THREE,4,...] STICKS, ALL STICKS, IT

Referenced by the game as:

```
You can see some sticks of dynamite here.
You see nothing special about the stick of dynamite.
That's not something you can close.
You can't wear that!
```

Example:

```
Class   TNT
  with  name 'stick' 'sticks//p' 'of' 'dynamite' 'tnt',
        short_name "stick of dynamite",
        plural "sticks of dynamite";

TNT     ->;
TNT     ->;
```

More information in the DM: §29

Make the objects 'indistinguishable' by creating a **Class** definition which includes a **short**_name property (to tell the game how to name a single instance of the object) and a **plural** property (to tell the game how to name a group of these objects). Add the modifier **//p** to the plural dictionary word in the **name** property.

The player can’t refer to THEM when they are grouped.

Most verbs (except TAKE, DROP) produce: "You can’t use multiple objects with that verb." if the player refers to more than one stick.

Enumerated set of indistinguishable nouns #2

For example: gold coins

Referenced by the player as: THE COIN, TEN [100,1000,...] COINS, ALL COINS, THEM

Referenced by the game as:

```
You can see one thousand gold coins here.
You see nothing special about the gold coins.
They're not something you can close.
You can't wear those!
```

Example:

More information in the DM: §3.11

```
Class Money(10)
 with name 'money' 'cash' 'gold',
      sing_name 'coin' 'dollar' 'zorkmid',
      plur_name 'coins' 'dollars' 'zorkmids',
      parse_name [ wd qty b p s;
        qty = TryNumber(wn);
        if (qty >= 0) { b++; wn++; }
        wd = NextWord();
        while (wd) {
          if (WordInProperty(wd,self,name)) b++;
          else
            if (WordInProperty(wd,self,plur_name)) p++;
            else
              if (WordInProperty(wd,self,sing_name)) s++;
              else break;
          wd = NextWord();
        }
        if (qty < 0)  ! TryNumber() didn't find a number
          if (s > p) qty = 1;
          else       qty = self.current_qty;
        if (p+s) self.request_qty = qty;
        return b+p+s;
      ],
      current_qty 0,  ! Number of coins in this object
      request_qty 0,  ! Number requested by player
      article [;
        if (self.current_qty == 1) print "a single";
        else        print (number) self.current_qty;
        rtrue;
      ],
      short_name [;
        if (self.current_qty == 1) print "gold coin";
        else                       print "gold coins";
        rtrue;
      ],
      before [ o;     ! Before any cmd for these coins
        if (self.request_qty == 0) self.request_qty = self.current_qty;
        if (self.request_qty > self.current_qty)
          return L__M(##Miscellany,42,self.current_qty);
        if (self.request_qty < self.current_qty) {
          o = self.current_qty - self.request_qty;
          o = Money.create(o, parent(self));
          if (o == nothing) "BUG: Money creation failed!";
          self.current_qty = self.request_qty;
          self.request_qty = 0;
        }
      ],
      each_turn [ o;
        self.request_qty = 0;
        objectloop (o ofclass Money && parent(o) ~= Money && o ~= self)
          if (parent(o) == parent(self)) {
            self.current_qty = self.current_qty + o.current_qty;
            Money.destroy(o);
          }
      ],
      adjust [ qty;
        if (qty >= 0) self.current_qty = self.current_qty + qty;
        else {
          qty = -qty;
          if (qty < self.current_qty)
            self.current_qty = self.current_qty - qty;
          else Money.destroy(self);
        }
      ],
      create [ qty loc;
        self.current_qty = qty;
        self.request_qty = 0;
        if (loc) move self to loc;
      ],
 has   pluralname;
```

The previous approach works well for relatively small numbers of indistinguishable items (say, fewer than 50), but because each item is an individual object, doesn't perform well for larger quantities. In a situation where the player has, for example, 1000 coins which he can spent, give to beggars, throw into fountains, and so on, you need something different (and more complex). This example uses a single object to represent all 1000 coins, which is fast and economical of Z-machine memory but causes some other problems: basically, you need to do a lot of the work that in the previous approach was done automatically by the library.

We supplement the standard **name** property (which contains 'neutral' words) with two of our own: **sing_name** and **plur_name**. Then, we use a **parse_name** routine to match against all of those, and also to look for a possible number typed first (TEN DOLLARS, 100 GOLD COINS); the routine stores such a value in the **request_qty** property as well as returning the number of matched words in the usual way. If no explicit number is typed, **request_qty** is set to either 1 (if the player typed just COIN), or -- if he typed COINS -- to the total number of coins available (which is stored in **current_qty**). We also need our own **article** and **short_name** properties, to handle the value in **current_qty**.

Having a single object for all 1000 coins is fine... until the player decides to DROP 50 COINS. At that point we need *two* objects: one representing the 50 coins on the ground, the other representing the 950 coins which the player is still holding. So, our **before** property creates a new object of the Money class, and adjust the **current_qty** properties of both objects. Then, the action continues... but only affects the object containing the specified quantity. The new object, respresenting the remainder of the coins, stays in the player's possession.

So far, so good... but what if the player decides to TAKE 50 COINS back into his inventory, or to DROP 100 COINS next to the original 50? We've now got two Money objects, where we want only one. The **each_turn** property takes care of this; it runs each turn looking for multiple Money objects with the same parent, and it combines them back into a single object.

Two final properties: **adjust** is provided so that other objects can add and subtract from **current_quantity**, destroying the Money object if that value reaches zero (**adjust** isn't used here, but might come into play if the player typed something like BUY BEANS FROM PEDDLER). And **create** brings a new Money object into being: we use it in the **before** property, and also in our **Initialise** routine -- the call is simply **Money.create(1000, player);** -- to give the player his initial stash of cash. Note that we have to use this technique, instead of defining an object's initial location within the object tree, because we are creating and destroying objects dynamically. The first line -- **Class Money(10)** -- says that up to ten separate collections of coins can be lying around at any one time; the actual number of coins in each of those collections is immaterial.

Grouped set of indistinguishable nouns

For example: fuses, grains of rice, rusty nails

Referenced by the player as:	THE FUSE, THE FUSES, IT

Referenced by the game as:	`You can see several fuses here.`
	`You see nothing special about the fuse.`
	`That's not something you can close.`
	`You can't wear that!`

Example:

More information in the DM: §27

```
Class   Fuse
  with  name 'fuse' 'fuses//p',
        short_name "fuse",
        list_together [;
            if (inventory_stage == 1) {
                print "several fuses";
                if (c_style & NEWLINE_BIT) new_line;
                rtrue;
                }
            ];
```

```
Fuse    ->;
Fuse    ->;
```

These are objects where two or more in the same place forms a group of indeterminate size. Define a **Class**, as in the previous example, and supply a **list_together** property with the listing rules you want to apply; this overrides the default library operation that would try to number how many instances are grouped together.

A terminating **new_line** is needed when the player's inventory includes "several fuses", but not when "several fuses" appear at the end of a room's description.

The player can’t refer to THEM when they are grouped.

Inexhaustible set, with ability to reference a single instance

For example: matches, pad of paper, bag of sweets

Referenced by the player as: THE MATCHES, THE MATCH

Referenced by the game as:

```
You can see some matches here.
You see nothing special about the matches.
They're not something you can close.
You can't wear those!
```

Example:

More information in the DM: §6 §20

```
Object  -> "matches"
  with  name 'matches',
        before [; Burn: <<Burn (child(self))>>; ],
  has   pluralname transparent;

Object  -> -> "match"
  with  name 'match',
        before [; PronounNotice(parent(self));
            Burn: return burning_match.light_me();
            default: <<(action) (parent(self)) second>>;
            ],
  has    pluralname;

Object  burning_match "burning match"
  with  name 'burning' 'lit' 'match',
        before [; Burn: "You've already lit one."; ],
        light_me [;
            if (parent(self)) <<Burn self>>;
            PronounNotice(self); move self to player;
            StartTimer(self, 4);
            "You light a match.";
            ],
        time_left 0,
        time_out [;
            remove self;
            "^The match burns to nothing.";
            ];
```

These are objects which exist as a group of infinite size, from which at any time no more than one instance can be detached: references to a single instance are normally taken as referring to the whole group, except where the action makes more sense applied to the individual. For example, TAKE MATCH, DROP PAPER and EXAMINE SWEET might apply to the group, while LIGHT MATCH, FOLD PAPER and UNWRAP SWEET might be more applicable to a single instance.

A single-instance object is defined as a child of the group object, which is given the **transparent** attribute to bring the child into scope (though *not* into view). The child delegates all actions other than Burn up to its parent, while the parent delegates a Burn action down to the child.

The act of lighting a match brings a new **burning_match** object temporarily into play; the processing is handled by that object's own **light_me** property, thus encapsulating the nitty-gritty details within the one place.

Virtual nouns

For example: cage of birds, bowl of fish, shopping list

Referenced by the player as: THE CAGE, THE BIRDS, IT

Referenced by the game as:

```
You can see a cage of birds here.
You see nothing special about the bird cage.
That's not something you can close.
You can't wear that!
```

Example:

More information in the DM: §26

```
Object  ->
  with  name 'bird' 'birds' 'cage',
        short_name [;
            if (self.number) print "cage of birds";
            else             print "bird cage";
            rtrue;
            ],
        invent [;
            if (inventory_stage == 1)
              switch (self.number) {
                0: print "an empty bird cage";
                1: print "a cage containing a bird";
                2: print "a pair of birds in a cage";
                default: print "a cage full of birds";
                }
            rtrue;
            ],
        number 2;
```

Occasionally you encounter a set of items, variable in number, which can't exist in isolation, only in the context of some other object. To avoid coding lots of objects, you might be able to simply instantiate the items by using their parent's **short_name** and **invent** properties.

Most of these examples are actually fairly simplistic, with even the relatively intricate Gold Coins object having some gaps in its implementation. There are often object-specific complexities -- typically related to manipulating them singly, in combination and as a group -- which mean that significant additional coding is needed. If you are trying to model real-world handling of a tricky subject like, for example, containers of liquid, you will need considerable patience to get things working properly; trial and error is usually the best approach.

Other examples of plural objects include: a pair of gloves (DM Exercise 14); the Scrabble pieces (DM Exercise 67); various coins (DM Exercises 68 and 69, also `Balances.inf`); and a book of matches, a box of candles and a pile of blocks (all from `Toyshop.inf`).

Why do my pronouns keep changing?

The library manipulates four pronouns -- IT, HIM, HER and THEM -- on the player's behalf, so that the most recently-mentioned objects match the appropriate pronouns. So far, so good: if you EXAMINE THE GRUMPY OLD MAN, that assigns the HIM pronoun to mean the old_man object; GIVE AXE TO HIM makes IT refer to the axe, and so on. You can use the PRONOUNS verb to check this out; the game's behaviour is nicely predictable.

Where, arguably, things start to fall apart is having the library adjust the pronoun assignments without your knowledge. After you type INVENTORY, then typically IT now refers to the *last* item that the player is carrying; after you type LOOK then typically IT refers to the item listed *last* in the room's contents. The other pronouns are similarly adjusted if there are NPCs or plural objects present. This can be highly confusing, especially since a TAKEn object appears *first* in the inventory, and a DROPped one *first* in the room description:

```
You can see a rusty axe, a bomb, some cartridges, a dagger and an
executioner here.

>X AXE
You see nothing special about the rusty axe.

>TAKE IT
Taken.

>INV
You are carrying:
  a rusty axe
  a shiny sword
  some grenades
  a nail-studded club

>X IT
You see nothing special about the nail-studded club.
```

To many players, this re-assignment of the settings can seem arbitrary and unpredictable, causing them to avoid the use of pronouns altogether. So, I recommend that you turn off the pronoun checking performed by INV and LOOK. It's easy to do; just define a **MANUAL_PRONOUNS** constant at the start of your game:

```
Constant Story "MYGAME";
Constant Headline "^My first Inform game.^";
Constant MANUAL_PRONOUNS;
```

Pronouns are now assigned only when the player refers directly to an object. Of course, this may mean that, as you move around, the object referenced by a pronoun is no longer in scope, but at least you can understand what's going on.

How do I define a new object property?

There are two ways: using *common* properties (rarely found), and using *individual* properties (very commonly found).

You can define a *common* property, one that is possessed by every object in the game, by adding a line like this near the start of the game:

More information in the DM: §3.5 §3.13

```
Property weight 0;
```

There's an Inform limit of 62 common properties in total, and the Library already defines 50 of them. This isn't actually much of a restriction, because it's quite rare to need common properties; most of the time, you can do the job better using individual properties.

You can define an *individual* property which applies only to one object, or to a class of objects, simply by mentioning its name in the **with** segment when defining the object or class; you don't need to use the **Property** directive. For example, you might say:

```
Object  old_man "grumpy old man"
  with  name 'man' 'grumpy' 'cross' 'old' 'elderly' 'ancient' 'gnome',
        grumpiness 3,
        ...
  has   animate;
```

in order to vary the old man's behaviour according to how grumpy he was at that moment. Perhaps his **description** property then becomes:

```
          description [;
              print "The grumpy old man ";
              switch(self.grumpiness) {
                  0:  "smiles bashfully.";
                  1:  "stares back without emotion.";
                  2:  "frowns in your direction.";
                  3:  "glowers at you ferociously.";
                  4:  "seems incandescent with rage.";
                  default: "stands silently in the corner.";
              } ],
```

In this example, the new **grumpiness** property contains a decimal value 0, 1, 2... etc. You can, however, assign other types of data to your property, as we explain next. Furthermore, you can assign more than one value to a property, thus turning it into an array. The syntax is exactly the same, except that you supply several values rather than just one. The familiar **name** property works just like this; here's another example, showing an alternative way of handling the **grumpiness** descriptions (though it would be sensible to include a check that **grumpiness** lay in the range 0..5):

```
  description [;
       print_ret "The grumpy old man ",
       (string) self.&grumpydesc-->(self.grumpiness);
      ],
  grumpydesc "smiles bashfully." "stares back without emotion."
      "frowns in your direction." "glowers at you ferociously."
      "seems incandescent with rage." "stands silently in the corner.",
```

More information in the DM: §3.5

How do I use my new object property?

You talk about it in the same way as the standard properties pre-defined by the library: that is, you'd refer to the previous topic's **grumpiness** property as **self.grumpiness** *within* the old_man object, or as **old_man.grumpiness** from within any other object. It's simple, consistent, and very powerful.

More information in the DM: §3.9

As for how you use your new property, that depends on what you store in it -- a true/false flag, a number, an object's address, a string or a routine (or an array of any of those, but we'll concentrate here on the basics). For example, you might create a **flammable** property, controlling what happens to an object when you set fire to it, as any of these formats:

Property definition	Reference to the property	What's the outcome?
`flammable,`	X = self.flammable;	X contains zero.
`flammable 20,`	X = self.flammable;	X contains 20.
flammable *obj_name*,	X = self.flammable;	X contains the address of the *obj_name*.
flammable "*string*",	X = self.flammable;	X contains the address of the string.
	X = self.flammable();	the string is printed; X contains **true**.
flammable [; *statement*; *statement*; ...],	X = self.flammable;	X contains the address of the routine.
	X = self.flammable();	the routine is executed; X contains its return value.

The really interesting -- and rather surprising -- information in that table relates to the **self.flammable()** format. Read the table again: you'll see that *the same format* does something useful both when the property value is a routine (which you'd expect), or a string (which you certainly wouldn't expect). This is a great feature (and is discussed in more detail later)!

It's often useful, especially when creating your own object classes, to construct properties which can be either a string or a routine. The library has several of these: **article**, **cant_go**, **description**, **each_turn**, **initial**, **inside_description**, **plural**, **when_XXX** and all of the direction properties. They're handy because they cater both for the simple case -- when you always want to display the same information -- and for more complex requirements where what you print depends on the object state, or the time of day, or some other factor.

Suppose that you've got an object, such as a flashlight, which can be pointed at other objects. Most objects don't react, but a few do... those that have a **when_lit** property, which could be a routine or string. You can choose one of three techniques.

Technique	For example	Commentary

Testing the property type yourself	`if (otherObject.when_lit ofclass Routine)`
	`    otherObject.when_lit();`
	`else`
	`    print (string) otherObject.when_lit;`
		Not recommended -- it's too easy to make a mistake.

Using a library routine	`PrintOrRun(otherObject, when_lit);`
	`PrintOrRun(otherObject, when lit, true);`
		Simpler and more reliable.
		In the first example a newline is automatically added after a printed string; in the second example this is suppressed.

Letting the system handle it automatically	`otherObject.when_lit();`
	`otherObject.when_lit(p1,p2, ...);`
		Simplest of all.
		A newline is *always* added after a printed string.
		In the second example, the parameters are available to the **when_lit** routine.

To help clarify what's happening, here's some actual code:

```
Object  -> "flashlight"
  with  name 'flashlight',
        before [;
          PointAt:
            if (self has light && second provides when_lit)
               return second.when_lit();
        ],
        after [;
          SwitchOn:  give self light;
          SwitchOff: give self ~light;
        ],
  has   switchable ~on ~light;

Object  -> "owl"
  with  name 'owl',
        when_lit "The owl blinks in mild surprise.",
  has   animate;

Object  -> "mouse"
  with  name 'mouse',
        when_lit [;
            remove self;
            "The startled mouse disappears into the undergrowth.";
        ],
  has   animate;

[ PointAtSub; "Pointless."; ];
```

```
Verb 'aim' 'point'
    * held 'at'/'on'/'towards' noun -> PointAt;

Extend only 'shine'
    * held 'at'/'on'/'towards' noun -> PointAt;
```

So, if you're coding a property whose value can be a routine or a string -- and, given it's so easy, why not? -- my advice is to invoke it with the ***obj.prop*()** syntax at all times *unless* you especially don't want that automatic newline afterwards, when you should instead use **PrintOrRun(*obj*, *prop*, true)**.

Do I need to understand private properties?

No. They offer no meaningful advantage over regular properties, and you can forget all about them.

More information in the DM: §3.6

How do I define a new object attribute?

Unlike properties, you can't define an attribute which applies only to one object, or to a class of objects. Inform's attributes are globally-defined, and are always associated with all objects in the game. You would create a new attribute by saying, near the start of the game, something like:

```
Attribute large;
```

and then including:

```
  has   large;
```

More information in the DM: §3.7

in the definition of objects which were, for example, too bulky to be pushed through a crack, or hidden in a rabbit hole, or whatever your game demanded.

There's an Inform limit of 48 attributes in total, and the Library already defines 31 of them. This, too, isn't actually much of a restriction, because it's quite rare to need new attributes; most of the time, you can do the job better using individual properties, as we explain next.

How can I use individual properties as attributes?

Attributes are variables which may have only two states: they're either present, or they're absent. These variables -- usually called 'flags' in programming jargon (the flag is either 'up' or 'down', meaning: set or clear, on or off, true or false, yes or no...) -- become really useful when you want to work with binary states: is the chest open, or closed? Has the banana been peeled, or not? etc.

More information in the DM: §3.7

The use of attributes is very easy and straightforward. As we have seen in the previous topic, you can create a new attribute with the directive:

```
Attribute peeled;
```

And then you can check whether an object has the attribute set (or *not* set) with:

```
if (banana has peeled) ...
if (banana hasnt peeled) ...
```

You can set the attribute with:

```
give banana peeled;
```

and clear it with:

```
  give banana ~peeled;
```

Inform, however, has a limited stock of attributes; you can create only about 17 additional customised flags. Moreover, once you define an attribute, you do it for all the objects in the game; by default, the new attribute is not set for any of the objects, but nevertheless it feels a bit awkward to have the possibility of, for example, a peeled television set.

Individual properties are variables which may hold much more than just two states, so they can function as flags with room to spare. An individual property is defined within one object and it affects only that object: if you have a banana and an orange, you'll have to define a **peeled** property for each (or maybe create a Fruit class -- see below -- from which they both inherit such a property). The syntax is a little bit more complex, but hardly a nuisance. To define a new individual property to act as flag, simply write:

```
  Object  banana "banana" fruitbowl
    with  name 'banana',
          peeled false,
          ...
```

More information in the DM: §3.5 §3.13

Now you can test if the banana has been peeled (or not) with:

```
  if (banana.peeled == true) ...
  if (banana.peeled == false) ...
```

and change the state of the banana with:

```
  banana.peeled = true;
  banana.peeled = false;
```

Note the use of '**==**' to test the value of the property, and '**=**' to assign the value of the property.

If these tests or changes are made from *within* the banana object, we recommend the use of **self** instead of **banana**:

```
  self.peeled = true;
  if (self.peeled == false) ...
```

So what's so hot about this slightly more verbose method for flags? Three main reasons.

- There is virtually no limit to the number of individual properties that you can define (the DM claims a limit of 16,320).
- It's a cleaner programming style to define suitable variables only where they are needed.
- Being a 16-bit variable, a local property can represent more than just two states.

To illustrate this last point (admittedly, we're not talking about simple flags any more), suppose that you want to create a set of 'flaming torch' objects which can be unlit, alight or burnt-out (and therefore useless). One way is to define two new attributes:

```
  Attribute usable;
  Attribute burning;
```

and you could then test an object's state by:

```
  if (myObject has usable && myObject hasnt burning) ...     ! It's unlit
  if (myObject has usable && myObject has burning) ...       ! It's lit
  if (myObject hasnt usable) ...                             ! It's burnt-out
```

But that's a clumsy approach; a better way is to use a single **torch_state** property:

```
if (myObject provides torch_state && myObject.torch_state == 0) ! Unlit
if (myObject provides torch_state && myObject.torch_state == 1) ! Lit
if (myObject provides torch_state && myObject.torch_state == 2) ! Burnt-out
```

Note that we're testing whether `myObject provides torch_state`; this is unnecessary if we *know* that myObject is a torch and therefore will definitely have a **torch_state** property, but essential if myObject might be an ordinary object which didn't have such a property as part of its definition. Note also that the 'attribute' method doesn't enable you to deduce reliably whether any given object is in fact a torch or not, whereas with the 'local property' method it's easy:

```
if (myObject provides torch_state) ...            ! It's a torch of some sort
```

An even better way, incidentally, would be to define a Class of torch objects (more about Classes in the next topic):

```
Class  Torch
  with torch_state;

Torch  myObject
       ...

if (myObject ofclass Torch) ...                   ! It's a torch of some sort
if (myObject ofclass Torch && myObject.torch_state == 1) ...      ! It's lit
```

In general, we advise you to define new attributes *only* where they will apply to a considerable number of objects in your game. When the same state settings apply to only one or two objects, the use of local properties is just as easy, and a lot more powerful.

If your game needs a lot of flags to control the state of play and you really can't afford the memory consumed by making each one a Global variable or an individual property, you might be interested in the `newflags.h` library contribution, Fredrik Ramsberg's revision of Adam Cadre's `flags.h`. (But if you do, be sure to define some meaningful constant names, for example:

```
Constant FED_RACCOON = 0;       ! Has Beauford eaten the corn and fed
                                ! the raccoon in the proper order?
Constant BOOKED_PLANE = 1;      ! Has Hildegard booked her plane tickets
                                ! with the correct credit card?
...
Constant WEASEL_IN_PANTS = 53; ! Is the weasel hiding in the pantaloons?
```

rather than using raw numbers in your code.)

What does class inheritance do for me?

More information in the DM: §3.8

Defining your own classes, so that you can create objects which are instances of those classes, is an easy and yet powerful technique. Whenever you find yourself defining two or more objects which have some common characteristics, ask yourself whether they could possibly belong to the same class. The most obvious example is a class used for rooms. Almost every time that you add a new location to your game, you'll want it to have light. So you can readily create a basic template:

```
Class   Room
  with  description "A bare and featureless room.",
  has   light;
```

and then take that as the starting point for your new room objects:

```
Room    hallway "Dingy hall"
  with  description "Steps lead down into darkness.",
        d_to cellar;
```

For more information on objects, classes and the use of properties and attributes, see Roger Firth's InFancy pages

```
Room    cellar "Cellar"
  with  description
            "Once the home of vintage wine; now only spiders remain."
        u_to hallway,
  has   ~light;
```

Using classes like this makes a game easier to write and easier to read, so it's worth mastering the technique early on.

There's no reason why an object can't be a member of two classes. For example, here's a class whose only purpose is to be tested against by the "snow" object:

```
Class   Snowy;

Object  "snow"
  with  name 'snow' 'drift' 'drifts',
        article "some",
        description "Pure white billowy drifts.",
        found_in [; return(location ofclass Snowy); ],
        before [; Take: "It's too fine and powdery."; ];
```

Given those definitions, you can conjure up the start of an icy landscape:

```
Room    northpole "The North Pole"
  class Snowy
  with  description "Windy, and white, and very very cold.",
        ...
```

When is a Dynamic class useful?

More information in the DM: §29

A class definition is a good way of handling similar objects, and an excellent way of dealing with identical objects. For example, this is all that's needed to put some eggs into a nest:

```
Object  -> "nest"
  with  name 'nest'
  has   container open;

Class   Egg
  with  short_name "egg",
        plural "eggs",
        name 'egg' 'eggs//p',
        description "Speckled blue and green.";

Egg     -> ->;
Egg     -> ->;
Egg     -> ->;
```

The game reports this as "You can see a nest (in which are three eggs) here". The "three eggs" are grouped because they're 'indistinguishable' -- they provide a **plural** property, and their **name** properties are the same.

Usually, this technique is all you need -- there are three eggs at the start and hopefully still three at the end -- but just occasionally you'll come across a situation where similar objects appear and disappear during the course of a game. If you find yourself needing to create identical objects as the game progresses, you might consider using a Dynamic class. We'll illustrate this by blowing bubbles (you'll have to imagine the wire loop and the soap solution). Here's a basic class:

```
Class   Bubble(10)
  with  short_name "bubble",
        plural "bubbles",
        name 'bubble' 'bubbles//p',
        description "The bubble floats gently in the air.";
```

That's hardly any different from the definition of the Egg class, is it? Just the **(10)** after the class name, which you should read as specifying that "no more than ten bubbles can exist at any one time". What *has* changed is the way that the bubbles are brought into existence. We'll conjure them up by enabling the player to type BLOW BUBBLE, using this extension of the library's standard BLOW verb:

More information in the DM: §3.11

```
[ BlowBubbleSub obj;
    obj = Bubble.create();
    if (obj == nothing)
        "Sorry, you've run out of bubbles.";
    move obj to location;
    "You carefully blow a bubble.";
];

Extend 'blow' first
       * 'bubble'/'bubbles'    -> BlowBubble;
```

The statement **obj = Bubble.create();** does the trick. Sending a **create** message to a Dynamic class definition brings a new object of that class into existence, and returns the number used to identify the object. The object is created in limbo -- without a parent -- so you'll usually need to move it into play; **move obj to location;** does that, and finally, you tell the player what's happened. If instead **Bubble.create()** returns **nothing** then the object couldn't be created (because you've already blown all of the specified 10), apologise to the player, and give up.

So far so good, but we can make a few improvements to our code. First, we simplify **BlowBubbleSub()** by making the newly-created object responsible for its own initialisation. We do this by adding a **create** property to the class:

```
Class   Bubble(10)
  with  short_name "bubble",
        plural "bubbles",
        name 'bubble' 'bubbles//p',
        description "The bubble floats gently in the air.",
        create [;
            move self to location;
            "You carefully blow a bubble.";
        ];

[ BlowBubbleSub;
    if (Bubble.create() == nothing)
        "You've run out of bubbles.";
];
```

The use of **Bubble.create()** tells the Bubble class to create a new Bubble object; the library contains all the code to do that. Then, the library calls the **create** property of the new object -- if it provides one -- to enable the object to set itself up: here, by moving itself to the current location and announcing its own birth.

The bubbles that we're creating here aren't very realistic, since they can be taken and dropped like solid objects. Let's make them burst if touched...

```
Class   Bubble(10)
  with  short_name "bubble",
        plural "bubbles",
        name 'bubble' 'bubbles//p',
        description "The bubble floats gently in the air.",
        create [;
            move self to location;
            "You carefully blow a bubble.";
        ],
        before [;
          Attack, Cut, Push, PushDir, Rub, Squeeze, Take,
          Taste, Tie, Touch, Turn, Wave:
            Bubble.destroy(self);
            "With a gentle 'pop', the bubble bursts.";
        ];
```

...or if deliberatly popped...

```
[ BurstSub;
    "You can't burst that!";
];

Verb 'burst' 'deflate' 'pop' 'prick' 'puncture'
    * multi                 -> Burst;
```

...or of their own accord after a few turns:

```
Class   Bubble(10)
  with  short_name "bubble",
        plural "bubbles",
        name 'bubble' 'bubbles//p',
        description "The bubble floats gently in the air.",
        create [;
            StartTimer(self, random(5));
            move self to location;
            "You carefully blow a bubble.";
        ],
        before [ player_gone;
          Attack, Burst, Cut, Push, PushDir, Rub, Squeeze,
          Take, Taste, Tie, Touch, Turn, Wave:
            StopTimer(self);
            player_gone = (self notin location);
            Bubble.destroy(self);
            if (player_gone) rtrue;
            "With a gentle 'pop', the bubble bursts.";
        ],
        time_left 0,
        time_out [; <<Burst self>>; ];

[ BlowBubbleSub;
    if (Bubble.create() == nothing)
        "You pause for a moment to get your breath back.";
];
```

Note how we handle the destruction of a burst bubble. Rather than just moving it out of sight by **remove self**, we use **Bubble.destroy(self)** which has the effect making the destroyed object re-usable by a subsequent call to **Bubble.create()**. We suppress the message about the bubble bursting if the player is no longer in the room. Once we've got ten bubbles in the air at the same time and so can't blow any more, we have only to wait until one of them bursts... and we can blow another one. We therefore change the message from **BlowBubbleSub()** to suggest that the ban on creation is a temporary limitation.

One final flourish: we'll stop the bubbles being indistinguishable by giving them some size and colour characteristics, chosen at random.

```
Class  Bubble(10)
  with short_name [;
         print (address) self.&name-->0, " ",
               (address) self.&name-->1, " bubble";
            rtrue;
        ],
       name '.size' '.colour' 'bubble' 'bubbles//p',
       description [; print_ret (The) self, " floats gently in the air."; ],
       create [;
            self.&name-->0 = random('tiny', 'small', 'regular', 'large',
                                    'enormous');
            self.&name-->1 = random('red','green','blue');
            PronounNotice(self);
            move self to location;
            StartTimer(self, random(5));
            "You carefully blow ", (a) self, ".";
        ],
        before [ player_gone s c;
          Attack, Burst, Cut, Push, PushDir, Rub, Squeeze,
          Take, Taste, Tie, Touch, Turn, Wave:
            StopTimer(self);
            player_gone = (self notin location);
            s = self.&name-->0; c = self.&name-->1;
            Bubble.destroy(self);
            self.&name-->0 = s; self.&name-->1 = c;
            if (player_gone) rtrue;
            "With a gentle 'pop', ", (the) self, " bursts.";
        ],
        time_left 0,
        time_out [; <<Burst self>>; ];
```

Several changes here. We've reserved two entries at the start of the **name** property to hold a pair of adjectives representing the bubble's size and colour; we write appropriate random values into these entries at the start of the **create** property. The **short_name** property uses the two adjectives to compose the object's name (for example, "tiny red bubble"). The various messages use print rules like "(The) self" in order to display that composite name. We've added **PronounNotice(self)** so that BLOW BUBBLE THEN BURST IT works as you'd imagine it should. And, somewhat obscurely, we remember the bubble's size and colour just before we destroy it, and reinstate them afterwards. Huh?

A bit of inside information may be helpful here. The declaration **Class Bubble(10) ... ;** actually compiles 11 identical bubble objects, and makes them all children of the Bubble class (which is also an object). The **Bubble.create()** call doesn't do much more than make one of the first ten of those child objects available to the game, and the **Bubble.destroy()** call just returns the 'destroyed' object to the pool of children. The eleventh object is never made available; it's there to provide a template so that, when an object is destroyed, its property values can be reset to their default state. Which means that the randomised size and colour values are lost, and you may see "You can't see 'it' (the .size .colour bubble) at the moment". That's why we reinstate the size and colour values after destroying a bubble. All of which means that our bubbles now behave something like this:

```
>BLOW BUBBLE
You carefully blow a small blue bubble.

>BLOW BUBBLE
You carefully blow a large red bubble.

>EXAMINE IT
The large red bubble floats gently in the air.

>BURST IT
With a gentle 'pop', the large red bubble bursts.

>EXAMINE IT
You can't see "it" (the large red bubble) at the moment.

>LOOK
...
You can see a small blue bubble here.
```

There's one other thing you should know. You might imagine that a statement like **objectloop (o ofclass Bubble) ...** would loop through your 'created' bubbles. Wrong: it loops through all 11 of the compiled bubbles. The statement you want is probably **objectloop (o ofclass Bubble && o notin Bubble) ...**

How can I reconfigure the Player Character (PC)?

In `Parserm.h`, the Library defines a standard object **selfobj** which represents the human person who is playing your game, and a variable **player** which normally contains the address of **selfobj**.

The INVENTORY command lists the objects which are the children of **player**, while EXAMINE ME invokes its **description** property. By default, that outputs "As good-looking as ever.", but you may well wish to change it; here are a couple of methods. The standard **selfobj** looks rather like this:

```
Object  selfobj "(self object)"
  with  short_name  [; return L__M(##Miscellany, 18); ],
        description [; return L__M(##Miscellany, 19); ],
        ...
  has   animate concealed proper transparent;
```

More information in the DM: §21

So, one way to change the descriptive text is to define a **LibraryMessages** object to intercept Miscellany message 19:

```
Include "Parser";

Object  LibraryMessages
  with  before [;
            Miscellany:
                if (lm_n == 19) "Not much change since you last looked.";
            ];

Include "VerbLib";
```

More information in the DM: §25

However, it's easier just to change the **description** property itself. For example, you could add this line to your **Initialise()** routine:

```
[ Initialise;
        player.description = "Not much change since you last looked.";
        ...
        ];
```

or this one, referencing the *address* of a suitable property routine:

```
[ PlayerDesc;
        if (children(self) == 0) "Forlorn and empty-handed.";
        "Your possessions don't inspire much confidence.";
        ];

[ Initialise;
        player.description = PlayerDesc;
        ...
        ];
```

Or, yet another possibility, you could supply your own player object, a modified version of the one in `Parserm.h`:

```
Object  mySelfobj "(self object)"
  with  short_name  [; return L__M(##Miscellany, 18); ],
        description [;
            if (children(self) == 0) "Forlorn and empty-handed.";
            "Your possessions don't inspire much confidence.";
            ],
        ...
  has   animate concealed proper transparent;

[ Initialise;
        player = mySelfobj;
        ...
        ];
```

More information in the DM: §A3

In the **Initialise()** routine, it's safe to assign object addresses directly to the **location** and **player** variables. Everywhere else, you should use the Library routines **PlayerTo()** and **ChangePlayer()** respectively.

Why does my game crash when I use move in an objectloop?

Usually, it doesn't nowadays, because many errors are caught by Strict checking. However, you need to be aware of the special-case handling of **objectloop**. If you use one of the three special forms (especially #1; the others are rarely seen), then your loop runs a lot faster, because the number of objects being tested is much smaller.

More information in the DM: §3.4

However, this optimisation is achieved by using the **sibling()** function to navigate the linked object tree, rather than simply iterating through all of the defined objects. And if you **move** or **remove** an object in the middle of an optimised loop, you'll screw up the linkage, and the results will be... interesting.

objectloop formats:

Standard

Code:

```
objectloop (condition) {
    statement;
    statement;
    ...
}
```

Generated code is equivalent to:

```
for (all objects in game) if (condition) {
      statement;
      statement;
      ...
  }
```

Special case #1 The direct children (but not grandchildren) of *object.*

Code:

```
objectloop (i in object) {
      statement;
      statement;
      ...
  }
```

Generated code is equivalent to:

```
for (i = child(object) : i : i = sibling(i)) {
      statement;
      statement;
      ...
  }
```

Special case #2 (rare) The children of *object's* parent (including *object* itself).

Code:

```
objectloop (i near object) {
      statement;
      statement;
      ...
  }
```

Generated code is equivalent to:

```
for  (i  =  child(parent(object))  :  i  :  i  =
sibling(i)) {
      statement;
      statement;
      ...
  }
```

Special case #3 (rare) *object* and its younger siblings.

Code:

Generated code is equivalent to:

```
objectloop (i from object) {
      statement;
      statement;
      ...
  }
for  (i  =  object  :  i  :  i  =
sibling(i)) {
      statement;
      statement;
```

```
      ...
  }
```

Using **move** and **remove** in 'standard' **objectloops** is fine. Also, the (*i* in *object*) form is 'special' only in that exact form -- if you add another condition like (*i* in *object* && *i* has *attribute*) or (*i* && *i* in *object*), it becomes safely 'standard'. Note: you can't add extra conditions to the other two special forms, which is probably why they're not often encountered.

Can I loop through *all* of an object's dependents?

You'll see from the previous topic that **objectloop (*i* in *object*)** selects the direct children of *object*, but not any other dependent objects -- grandchildren, great-grandchildren and so on -- lower down the hierarchy. Since that's quite a useful thing to be able to do, we'll discuss some ways that you can go about it. First of all, for reasons that will hopefully become clear in a minute, we'll encapsulate the actual processing in a separate routine: this simple example merely displays the name of each object as we loop through it:

```
[ ProcessObject obj;
    print (name) obj, "^";
];
```

With that in place, here are three alternative definitions of a routine which applies ProcessObject() to all of the dependents of a specified parent object *par*:

```
[ R1 par   o;
    objectloop (o ~= par && IndirectlyContains(par, o))
        ProcessObject(o);
];

[ R2 par   o;
    for (o=child(par) : o : ) {
        ProcessObject(o);
        if (child(o)) o = child(o);
        else
            while (o) {
                if (sibling(o)) { o = sibling(o); break; }
                o = parent(o);
                if (o == par) return;
            }
    }
];

[ R3 par   o;
    objectloop (o in par) {
        ProcessObject(o);
        if (child(o)) R3(o);
    }
];
```

The R1() example is the easiest to write and understand: unfortunately it's also hopelessly inefficient because of the amount of unnecessary testing that it performs. The much better R2() example calls the low-level child(), sibling() and parent() functions to move directly between the various dependents of *par*. And R3() is a nice compromise: it uses recursion -- that is, it calls itself -- to navigate through *par*'s dependents, which makes it appear simpler than it actually is.

So far, so good: you could use our little ProcessObject() routine with any of R1(player), R2(player) or R3(player) to print a list of the player's possessions. Or in fact, not so good, because ProcessObject() is being called for *all* of *par*'s dependents, even if they're not visible to the outside world. For example, if the player is carrying a locked box, then you quite possibly want to bypass any objects in that box, rather than call ProcessObject() for each of them. This is quite tricky to do using the R1() technique, but is very straightforward for R2() and R(3). First of all, we extend ProcessObject() so that it returns true if its contents, if any, are also to be processed, or false if any contents are to be skipped. Here's one test that you might find useful:

```
[ ProcessObject obj;
    print (name) obj, "^";
    if ((obj has supporter or transparent) ||
         (obj has container && obj has open)) rtrue;
    rfalse;
];
```

And these are revised versions of R2() and R3() which test the value returned by ProcessObject() and act accordingly:

```
[ R2 par   o;
    for (o=child(par) : o : ) {
        if (ProcessObject(o) && child(o)) o = child(o);
        else
            while (o) {
                if (sibling(o)) { o = sibling(o); break; }
                o = parent(o);
                if (o == par) return;
            }
    }
];

[ R3 par   o;
    objectloop (o in par)
        if (ProcessObject(o) && child(o)) R3(o);
];
```

6 · Verbal Versatility

How do I define a new verb?

Whereas new nouns and adjectives are defined throughout the game file -- more or less whenever you create an object -- the process of adding new verbs needs more careful consideration. It's a complex subject, so we'll just scratch the surface. Basically, you need to define the verb's grammar -- what word(s) the player is permitted to type -- and an action routine which is invoked when he actually does type them. The simplest case is an intransitive verb; one that doesn't take an object. For example, to enable the player to SMILE and FROWN, you might add these lines at the end of your game, after the **Include "Grammar"**:

```
[ SmileSub; "Your face lights up in a cheery smile."; ];
[ FrownSub; "A brief hint of annoyance crosses your face."; ];

Verb 'smile' 'grin' 'smirk' 'beam' 'twinkle'
    *           -> Smile;
Verb 'frown' 'scowl' 'glower' 'glare' 'lour'
    *           -> Frown;
```

More information in the DM: §30 §31

Here, 'smile' and the other words in apostrophes are the verbs being added to the dictionary; since you can't be sure exactly which word the player might think of, you can collect several verbs which are roughly similar in effect, and make them all behave the same. The blank space within "* ->" means that you'll recognise SMILE only on its own, not when it's followed by another word. The "Smile" after the "->" is the name of the action to be performed when you recognise SMILE. And finally, Inform automatically derives the name of the routine called to deal with an action by appending "Sub" to the name of the action. So, SmileSub is the routine which is called to deal with the Smile action.

The player can now express some basic facial emotion to the world in general. But what if he wants to SMILE AT something or someone? Here's a revised grammar:

```
Verb 'smile' 'grin' 'smirk' 'beam' 'twinkle'
    *           -> Smile
    * 'at' noun -> Smile;
```

And here's an enhanced action routine, generating responses which are better-tailored to the subject:

```
[ SmileSub;
    if (noun == nothing) "Your face lights up in a cheery smile.";
    if (noun == player) "Easily amused, heh?";
    if (noun has animate) print_ret (The) noun, " looks at you quizzically.";
    "There is no reaction.";
    ];
```

When writing an action routine -- FrownSub, SmileSub, etc -- you should generally provide just a standard default response which can safely be used in the majority of circumstances; that is, don't try to handle special cases -- ones that actually affect the game world -- in the action routine. Then you'll use **before** (and occasionally **after**) properties to define the behaviour of the handful of objects, often only one, which need to respond in a specific way.

When should I use before/react_before properties?

You use a **before** property to intercept an action which is aimed specifically at that object; you use a **react_before** property to intercept an action which is aimed at another, nearby, object. For example, if you introduce the grumpy old man into the room, you can SMILE AT MAN to elicit a quizzical glance in return. Or, with a suitable **before** property, he can be taught to respond differently:

```
Object  old_man "grumpy old man" study
  with  name 'man' 'grumpy' 'cross' 'old' 'elderly' 'ancient' 'gnome',
        description "The grumpy old man stands silently in the corner.",
        before [; Smile: "The grumpy old man nods slightly."; ],
  has   animate;
```

To make him respond when you SMILE AT DESK, or at something else in the room, just provide a **react_before** property:

More information in the DM: §4 §6 §8 §17

```
Object  old_man "grumpy old man" study
  with  name 'man' 'grumpy' 'cross' 'old' 'elderly' 'ancient' 'gnome',
        description "The grumpy old man stands silently in the corner.",
        before [; Smile: "The grumpy old man nods slightly."; ],
        react_before [; Smile: "The grumpy old man looks puzzled."; ],
  has   animate;
```

Actually, that's not quite right. The old man is now looking puzzled too often; his **react_before** is intercepting *every* SMILE in the vicinity, even ones aimed at him. You'd be better using:

```
        react_before [; Smile: if (noun == writing_desk)
                                    "The grumpy old man looks puzzled."; ],
```

or perhaps:

```
        react_before [; Smile: if (noun ~= self or player or nothing)
                                    "The grumpy old man looks puzzled."; ],
```

Understanding the power of the **before** (and to a lesser extent, **react_before**) properties is the key to making specific objects respond to specific actions. There are also **after** and **react_after** properties, but you don't use them quite so often, because they apply only to Group 2 actions -- those like Take and Open and Insert which change the game's state. Actions like Push or Show, or Smile, are in Group 3; they don't change the state unless by supplying a suitable **before** property you cause them to do so. For example, you'd be wasting your time adding **after**/**react_after** properties for Smile to the old man -- they'd simply never be invoked by this example code.

A slightly simplified list of the order in which a command like DOTHIS TOTHAT is processed is:

More information in the DM: §6

"Before" stage	1. any **react_before** properties of objects in scope, including **player** and TOTHAT 2. any **before** property of the room 3. any **before** property of TOTHAT 4. any **life** property of TOTHAT (but only if it's **animate**) -- see the next topic
"During" stage	5. the DOTHISSub action routine (which is generally buried deep in the Library)
"After" stage (only for Group 2 actions)	6. any **react_after** properties of objects in scope, including **player** and TOTHAT 7. any **after** property of the room 8. any **after** property of TOTHAT

If at any stage the routine doing that step's processing returns **true**, the whole sequence ends immediately. So, for example, when the old man's **react_before** printed "The grumpy old man looks puzzled." and returned **true** at step 1, none of the other steps took place.

Where do life and orders fit in?

Well, unless an object has an **animate** attribute, the **life** and **orders** properties don't have any effect at all (except that if an object has a **talkable** attribute, you can give it a **life** property). If your object *is* **animate**, there are four cases to consider; note that these processing sequences focus on the objects being addressed, and ignore any reactions of other objects:

Case	**Example**	**Processing sequence**
Primary object	ATTACK OLD MAN	1. any **before** property of **old_man** (test for **##Attack** action) 2. any **life** property of **old_man** (test for **##Attack** action) 3. the **AttackSub** action routine
Secondary object (with THROW)	THROW AXE AT MAN	1. any **before** property of **axe** (test for **##ThrowAt** action) 2. any **before** property of **old_man** (test for **##ThrownAt** fake action) 3. any **life** property of **old_man** (test for **##ThrowAt** action) [1] 4. the **ThrowAtSub** action routine
Secondary object (with GIVE/ SHOW)	GIVE AXE TO MAN	1. any **before** property of **axe** (test for **##Give** action) 2. any **life** property of **old_man** (test for **##Give** action) [2] 3. the **GiveSub** action routine
Direct order	MAN, TAKE THE AXE	1. any **orders** property of **old_man** (test for **##Take** action) 2. any **life** property of **old_man** (test for **##Order** fake action, then **##Take** action) [3] 3. parser displays "The grumpy old man has better things to do"

Note 1. Whereas the man's **before** property is passed the expected ##ThrownAt fake action (because the old man is the target of the axe throwing), his **life** property -- rather surprisingly -- is passed ##ThrowAt.

Note 2. If you were expecting the man's **before** property to be called with the ##Receive fake action before this step, you'd be wrong.

Note 3. This clumsy method is a hangover from the way orders were originally handled, and is best avoided. If your NPCs are intended to respond to direct commands, give them an **orders** property.

The bottom line is: **orders** is pretty well essential if your NPCs can expect to be requested to perform actions. On the other hand, **life** isn't so useful; just about its only real advantages over the more general **before** are that it provides a common point at which to reject the nine animate-only actions (Attack, Kiss, WakeOther, ThrowAt, Give, Show, Ask, Tell and Answer), and that it seems to be the only place where you can customize responses to GIVE and SHOW.

Surely the syntax of these properties is a little odd?

Yes. When you write something like:

```
before [;
    Smile: "The grumpy old man nods slightly.";
    Touch: "Be gentle!";
    Smell: "Tobacco and old boots.";
],
```

it's as though you'd written a **switch** statement, omitting the highlighted material:

```
before [;
  switch(sw__var) {
    ##Smile: "The grumpy old man nods slightly.";
    ##Touch: "Be gentle!";
    ##Smell: "Tobacco and old boots.";
  }
],
```

(The implicit **switch** statement is testing the library variable **sw__var**, which usually contains the current **action** value, except in **life** properties where it contains **reason_code**.) Because **before** (and in fact all properties) pseudo-conforms to the **switch** syntax, you can add code in two places for special effect:

```
        before [;
            ! statements here are executed for all actions.
            Smile: "The grumpy old man nods slightly.";
            Touch: "Be gentle!";
            Smell: "Tobacco and old boots.";
            default: ! statements here are executed for non-listed actions.
            ],
```

How do I change an existing verb?

As you've seen from the earlier topic, the process for creating a brand new verb -- one which the Library doesn't already define -- is reasonably straightforward. Here's another example, perhaps slightly more complex, of verbs which interact with objects and make changes in the model world. Our aim is to create a grammar enabling the player to type FOLD PAPER ROUND BOOK or WRAP BOOK IN PAPER:

More information in the DM: §30 §31

```
[ CoverWithSub; "That's hardly a suitable covering!"; ];

Verb 'wrap' 'fold' 'enfold'
    * noun 'round'/'around'/'over' held     -> CoverWith
    * held 'in'/'inside'/'with' noun        -> CoverWith reverse;
```

This time, two objects are involved: the wrapper, and something in the player's possession which is being wrapped. Let's try and write this fairly generally (so that *any* held object can be parcelled up) by aiming the **CoverWith** action at the wrapper object, which then needs a **before** property to do the real work:

```
! The 'paper' object is in the same room as the object to be wrapped.

Object  -> paper "sheet of wrapping paper"
  with  name 'sheet' 'of' 'wrapping' 'paper',
        before [; CoverWith:
            move parcel to parent(self);
            move second to parcel; move self to parcel;
            "You wrap ", (the) self, " around ", (the) second, ".";
            ];

! Initially, the 'parcel' object has no parent

Object  parcel "folded paper parcel"
  with  name 'folded' 'paper' 'parcel' 'package',
        before [; Open:
            while (child(self)) move child(self) to parent(self);
            remove self;
            "You unwrap ", (the) self, ".";
            ];
```

So far, so good: our new WRAP verb brings the parcel object into the room, with the paper and the original unwrapped object as children, and the existing OPEN/UNWRAP verbs turn that parcel back into its wrapper and content objects. It all works... providing the player thinks to WRAP (or FOLD or ENFOLD) the paper round the book, or WRAP the book in paper. But of course he *doesn't* think of that, at least initially. Not being able to Read The Author's Mind, the first thing our player tries is PUT PAPER AROUND BOOK, followed by COVER BOOK WITH PAPER. Since a fundamental principle of good game design is to respond intelligently to just about any reasonable player command, we need to accomodate those forms as well; that's where things get... interesting. We can't just include 'put' 'cover' in the new grammar after 'wrap' 'fold' 'enfold', because both 'put' and 'cover' are already defined in the Library file `Grammar.h` (as soon as you start thinking about extensions to Inform's standard list of verbs, you'll need to study the contents of that file):

```
Verb 'close' 'shut' 'cover'
    * noun                                  -> Close
    * 'up' noun                             -> Close
    * 'off' noun                            -> SwitchOff;
```

```
Verb 'put'
    * multiexcept 'in'/'inside'/'into' noun -> Insert
    * multiexcept 'on'/'onto' noun          -> PutOn
    * 'on' held                             -> Wear
    * 'down' multiheld                      -> Drop
    * multiheld 'down'                      -> Drop;
```

So, we need to modify the existing grammar to suit our needs, and Inform provides the **Extend** directive for this purpose. If we simply want to add a line of grammar, we can code:

```
Extend 'put' last
    * noun 'round'/'around'/'over' held     -> CoverWith;
```

The keyword **last** places the additional grammar after the existing definitions. In fact, that's the default: you can omit **last** if you want the new grammar at the end, or you can use instead **first** to insert it at the start, or **replace** to override the existing grammar completely. It's worth noting here that the debugging tool SHOWVERB is very useful once you find yourself playing with the verb grammars:

```
>SHOWVERB PUT
Verb 'put'
     * multiexcept 'in' / 'inside' / 'into' noun -> Insert
     * multiexcept 'on' / 'onto' noun -> PutOn
     * 'on' held -> Wear
     * 'down' multiheld -> Drop
     * multiheld 'down' -> Drop
     * noun 'round' / 'around' / 'over' held -> CoverWith
```

That's extended the existing PUT verb satisfactorily, but COVER needs a little more care. If we code it like this:

```
Extend 'cover'
  * held 'in'/'with' noun                   -> CoverWith reverse;
```

then we're actually extending CLOSE and SHUT as well, as we can see:

```
>SHOWVERB COVER
Verb 'close' 'cover' 'shut'
     * noun -> Close
     * 'up' noun -> Close
     * 'off' noun -> SwitchOff
     * held 'in' / 'with' noun -> CoverWith reverse
```

Now the player can type CLOSE BOOK WITH PAPER, a most unsatisfactory side effect. So this is where another **Extend** keyword -- **only** -- comes into play. To Extend the existing grammar only for the verb 'cover' -- but not for 'close' or 'shut' -- we code thus:

```
Extend only 'cover'
    * held 'in'/'with' noun                 -> CoverWith reverse;
```

And this adjusts the grammar in the desired manner:

```
>SHOWVERB COVER
Verb 'cover'
     * noun -> Close
     * 'up' noun -> Close
     * 'off' noun -> SwitchOff
     * held 'in' / 'with' noun -> CoverWith reverse

>SHOWVERB CLOSE
Verb 'close' 'shut'
     * noun -> Close
     * 'up' noun -> Close
     * 'off' noun -> SwitchOff
```

Finally, just for completeness, we'll define 'unfold' as a synonym for 'open', so that we can UNFOLD PARCEL as well. This is very easy to do:

```
Verb 'unfold' = 'open';
```

Now let's see another example to explore the versatility of the **Extend** directive. The Library defines this simple drinking grammar, in which 'drink', 'swallow' and 'sip' all behave identically:

```
Verb 'drink' 'swallow' 'sip'
      * noun                                -> Drink;
```

Imagine that you want to distinguish between sipping (a small quantity), drinking (a normal quantity) and swallowing (the whole lot), at the same time adding a few synonyms. Three new action routines must take the place of the standard **DrinkSub()**:

```
[ SipSub;     L__M(##Drink, 1, noun); ];
[ SwigSub;    L__M(##Drink, 1, noun); ];
[ SwallowSub; L__M(##Drink, 1, noun); ];
```

These new routines are defined so that they all provide by default the standard library message that was intended for the verb 'drink'. We now code synonyms for 'drink':

```
Verb 'imbibe' 'swig' 'gulp' 'quaff' = 'drink';
```

And lastly, we **Extend** the grammar so that the verbs are correctly redirected to the new action routines:

```
Extend only 'sip' replace
    * noun          -> Sip;

Extend only 'drink' 'imbibe' 'swig' replace
    * noun          -> Swig
    * 'all' noun    -> Swallow;

Extend only 'swallow' 'gulp' 'quaff' replace
    * noun          -> Swallow
    * 'all' noun    -> Swallow;
```

The **only** keyword ensures that the redirection applies just to the listed verbs. Here we have also added the **replace** keyword, which tells Inform to ignore completely the old grammar for these verbs and instead utilise the new one we have defined. Of course, your drinkable objects now need to handle the three actions Sip, Swig and Swallow instead of simply Drink; additional code is usually the price of thoroughness.

Can I remove an existing verb?

Now that we know how to add new verbs, and modify or extend existing ones, only one challenge remains: preventing Inform from recognising a verb that's already defined in `Grammar.h`. Of course, just editing it out of the file is one way, but as always it's safer to override the Library rather than change it. So, define these routines:

More information in the DM: §31

```
[ Anything; ! Ignore the remaining input line
    while (NextWordStopped() ~= -1);
    return GPR_PREPOSITION;
    ];

[ NoSuchVerbSub; L__M(##Miscellany, 38); ];
```

Having done that, you can **Extend...replace** the existing verbs that you wish to remove. For example, if Momma don't allow no swearwords used round here:

```
Extend 'shit' replace
    * Anything -> NoSuchVerb;

Extend 'bother' replace
    * Anything -> NoSuchVerb;
```

Why are actions labelled Group 1, Group 2 or Group 3?

The labelling is just a handy way of distinguishing between three types of behaviour:

More information in the DM: §6 Table 6

- **Group 1 actions** are things like SAVE, SCORE and the debugging verbs. They're called 'meta' actions because they control the game itself and -- unlike the other two groups -- don't increment the time and turns counters, nor otherwise affect the model world within the game. It's pretty unusual to create your own Group 1 actions, which you do by including the word **meta** after the **Verb** directive.
- **Group 2 actions** are 'active' things like TAKE, CLOSE, EXAMINE and INVENTORY. For all of these, the standard library action either changes the state of the model world, or displays some information about it. An object's **before** property can intercept one of these actions to substitute alternative behaviour for what the library would otherwise do, or simply to prohibit it completely in some or all circumstances. An object's **after** property can step in once the library action has happened, but before the player has been told; this is the point to add some supplementary behaviour, or to display a customised message to the player.
- **Group 3 actions** are 'passive' things like TOUCH, ATTACK, CLIMB and CUT. For all of these, the standard library action simply displays some message of bland refusal or neutral feedback; no change to the state of the model world occurs. As before, an object's **before** property can intercept one of these actions to supply a better message, or to substitute some more active behaviour. However, that's as far it it normally goes; because the library doesn't change the model world, there isn't an 'afterwards' that differs from the previous state, and so the library ignores an object's **after** property even if you provide one.

So, what makes an action 'Group 2' or 'Group 3'? It's pretty simple, really; it just comes down to what you write in the action routine. Here's a typical Group 2 action routine:

```
[ CloseSub;
    if (ObjectIsUntouchable(noun)) return;               ! Line 1
    if (noun hasnt openable) return L__M(##Close,1,noun); ! Line 2
    if (noun hasnt open)     return L__M(##Close,2,noun); ! Line 3
    give noun ~open;                                      ! Line 4
    if (AfterRoutines() || keep_silent) rtrue;            ! Line 5
    L__M(##Close,3,noun);                                 ! Line 6
];
```

- In Line 1, the action routine gives up immediately if the target object is out of reach and can't physically be closed; the **ObjectIsUntouchable()** routine will already have displayed an appropriate message, so **CloseSub()** can simply return.
- Lines 2 and 3 display appropriate messages if the target object can't logically be closed, using the **L__M()** routine to do the actual output.
- If we get as far as Line 4, the CLOSE is guaranteed to work, so we modify the state of the target object. That tiny attribute change is all that actually happens; it's only in your imagination that the heavy oak lid comes crashing down.
- Line 5 calls **AfterRoutines()**, which invokes the target object's **after** property (if it has one) and returns whatever that property returned. If that's **true**, the **after** property has already displayed a message, so we can finish without further ado. We also quit here if the library variable **keep_silent** has been set.
- If we get to Line 6, the action has been performed but the player doesn't yet know, so we use **L__M()** to display a confirmatory "You close the **...**"

Contrast that routine with this typical Group 3 action:

```
[ CutSub; L__M(##Cut,1,noun); ];
```

and you can see the difference: just a message; no state change, no call to **AfterRoutines()**. It doesn't matter what you try to CUT; nothing's going to change unless you make it happen.

So let's do just that -- turn **CutSub()** into a Group 2 routine. For compatibility with the current library behaviour, we want to be able to classify an object as a 'cutter' or 'cuttable' or -- usually -- neither. A successful CUT will require a 'cutter' working on a 'cuttable' (for example, CUT ICE WITH AXE); other combinations (such as CUT ICE WITH PENGUIN or CUT SNOWMOBILE WITH KNIFE) won't be allowed.

We'll use local property variables for the 'cutter' and 'cuttable' classification, like this:

```
Object  -> "hazel tree"
  with  name 'hazel' 'tree' 'branch' 'branches',
        description "The lower branches are firm and straight.",
        cuttable 1;

Object  -> "knife"
  with  name 'knife',
        description "Sharp enough for light wood-cutting.",
        cutter 1;
```

And then all we've got to do is to write a more powerful **CutSub()** to Replace the one in the library, and Extend the grammar a little. Here it all is:

```
Replace CutSub;

Include "VerbLib";
Include "Grammar";

[ CutSub x;
    if (~~(noun provides cuttable && noun.cuttable))        ! Line 1
        return L__M(##Cut,1,noun);                          ! Line 2
    if (second) {                                           ! Line 3
        if (~~(second provides cutter && second.cutter))    ! Line 4
            print_ret (The) second, " can't cut anything."; ! Line 5
    }                                                       ! Line 6
    else                                                    ! Line 7
        "You need to specify a sharp implement.";           ! Line 8
    if (AfterRoutines() || keep_silent) rtrue;              ! Line 9
    "You make a small incision in ", (the) noun, ".";       ! Line 10
];

Extend 'cut'
    * noun 'with' noun  -> Cut;
```

- Lines 1 and 2 test whether the target object is **cuttable**; most object aren't, and so we generate the conventional "Cutting that up would achieve little".
- Line 3 tests whether a second object (...WITH KNIFE) was mentioned.
- Lines 4 and 5 test whether the second object is a **cutter**; most object aren't, and so we say so.
- We get to Line 8 if there was no second object, so we remind players what they've omitted.
- By Line 9, we know we've got a cuttable and a cutter, so we're all systems go. The **AfterRoutines()** and **keep_silent** work exactly as before.
- If we get to Line 10, the action has been performed but the player doesn't yet know, so we display a confirmatory message.

That's a splendid stuff, except that you might notice one tiny omission: we haven't actually changed the state of the model world in any way (as we did with `give noun ~open;` in **CloseSub()** above). There's a good reason for this: the library doesn't include a **cut** attribute, and you can easily see why if you think of a few examples. Consider CUT CAKE, CUT CORN, CUT MYSELF, CUT PHONE LINE, CUT ROPE... the results are significantly different, and you usually need to do something more sophisticated than just setting or clearing an attribute, something specific to the object being cut.

How do we do this? the object's **after** property provides a perfect spot. By the time we call **AfterRoutines()** at Line 9, we know that the CUT action is theoretically possible; we can just let the **after** property decide whether to actually go ahead, and with what outcome. Ideally, the **after** should always return true, so our "incision" message should never appear in practice. Let's explain this with an example; we'll enhance our cuttable tree so that the player can hack off a walking staff (once only, but that should be sufficient).

```
Object  -> "hazel tree"
  with  name 'hazel' 'tree' 'branch' 'branches',
        description "The lower branches are firm and straight.",
        cuttable 1,
        after [;
          Cut:
            if (staff in self) {
                move staff to player;
                "You select a straight branch, neatly cut it from the tree,
                 and trim away the side twigs to form a stout walking
                 staff.";
            }
            else
                "It would be vandalism to hack off another branch.";
        ];

Object  -> -> staff "walking staff"
  with  name 'walking' 'staff' 'stick' 'pole',
        description
            "About five feet long, and a little thicker than your thumb.";
```

Initially, we hide the staff object as a child of the tree (which doesn't have **container** or **transparent** attributes, so the player won't know that it's there). When the player cuts the tree with his knife, the **after** property moves the staff into his possession, and the `if (staff in self) ...` test prevents any further tree surgery. By using similar principles, you should be able to handle most CUT X WITH Y requirements.

We're nearly done; just time for a couple of enhancements to our **CutSub()**. Since **cuttable** and **cutter** are actually variables, we can use them for more than simple true/false flags. In the following code, they're treated as a sort of Moh's Hardness Scale, wherein for example a **cuttable** value of 3 can only be cut by a **cutter** whose value is 3 or more, so that a penknife could cut paper, but not wood. The other enhancement is more intelligent handling of the situation where the player's CUT TREE omits the WITH KNIFE. A objectloop searches for a suitably powerful cutter in scope and, if it finds exactly one such implement, uses it by default. These additional features add only a few lines to the routine:

```
[ CutSub x;
    if (~~(noun provides cuttable && noun.cuttable))
        return L__M(##Cut,1,noun);
    if (second) {
        if (~~(second provides cutter && second.cutter))
            print_ret (The) second, " can't cut anything.";
        if (second.cutter < noun.cuttable)
            print_ret (The) second, " isn't sharp enough to cut the ", (the)
                       noun, ".";
    }
    else {
        objectloop (x provides cutter && TestScope(x) && x.cutter >=
                    noun.cuttable)
            if (second == nothing) second = x;
            else                   second = -1;
        if (second <= 0) "You need to specify a sharp implement.";
        print "(with ", (the) second, ")^";
    }
    if (AfterRoutines() || keep_silent) rtrue;
    "You make a small incision in ", (the) noun, ".";
];
```

How do I detect the player entering a room, or trying to leave?

You need to trap the "Go" action in **before** and **after** properties of rooms. A room's **before** routine is considered when the PC is trying to leave that room, while an **after** routine is triggered for the room where the PC arrives. In both cases **noun** is one of the direction objects (n_obj, s_obj, and so on) representing the direction of movement, so you can test for paths that need special handling; otherwise, your rules apply to all exits or entrances.

Before: The PC's action of moving in the desired direction triggers whatever special behaviour you have coded. If the routine returns **true**, the action is interrupted and the PC remains in the same location; if not, the normal movement rules apply. For example:

```
Object kitchen "Kitchen"
  with description
           "An old room, devoid of furniture. There's a big window set
            in the west wall.",
       before [; Go:
           if (noun == w_obj) {
               if (kitchen_window has open)
                   "The fall might be hazardous to your health.";
               "Bonk! You walk into the closed window.";
               }
           if (kitchen_window has open) {
               give kitchen_window ~open;
               print "Before you leave the kitchen, you close the window.^";
               }
           ],
       n_to corridor,
  has  light;
```

More information in the DM: §8

The lines concerning Westerly movement all return **true**, preventing the player from heading into oblivion, while those for other movements return **false**, thus permitting him to leave by any remaining exit, closing the window if necessary and printing an appropriate message on the way out.

After: The coded rules happen just after the PC arrives in the room, but before the room description is printed; if the routine returns **true**, no description appears on the screen. For example:

```
Object  traps_room "Traps Room"
  with  description
            "Yet another dangerous room in the Mad Overlord's castle.",
        after [; Go:
            if (noun == s_obj && black_door.snare == true) {
                print "Just as you enter the room, you hear a loud click
                       and watch in horror as tens of steel daggers fly in
                       your direction.^";
                deadflag = 1;
                rtrue;
                }
            if (noun == w_obj && children(player)>=3) {
                print "^Under the weight of your possessions,
                       the floor gives way and you fall into...^";
                PlayerTo(cellar);
                rtrue;
                }
            if (noun == u_obj) {
                PlayerTo(self);
                "[That was some nice climbing.]";
                }
            ],
        n_to passage,
        e_to throne_room,
  has   light;
```

Quite a few possibilities are going on in this example. The PC gets killed if he enters the room by heading South from the passage (and the trap of the door is set); note that we return **true**, so that the room description will not be printed after the "Just as you..." message. If the PC comes West from the throne_room with three or more possessions, he will fall into the cellar; the custom message "Under the weight..." is printed before the new room description (which, thanks to **PlayerTo()**, will be the cellar instead of the traps_room). Again, we return **true** to prevent the cellar's description being printed twice -- once for **PlayerTo()** and once for normal movement rules -- though this could also have been avoided by coding **PlayerTo(cellar,1)**. Finally, if the PC climbs back up from the cellar, we get another message, but this time it will be printed *after* the room description; note that the printing syntax is returning **true** by itself.

You may trap "Go" actions both in **before** and **after** properties of the same room. In the above example, you could easily provide a realistic touch (or a clue) to the falling trap by adding:

```
  before [; Go: if (noun == e_obj)
                 print "[stepping over the weak-looking section of floor]^";
         ],
```

It's useful to know the sequence of actions performed by the library when the player moves to a new location -- especially the order in which text is printed -- so that you may interrupt the process at will or include a message in the desired place. This list (which apart from the PC's movement is very similar to the sequence for a "Look" action) is a simplified model, ignoring the evaluation of light and the possibility of the player being in a container or on a supporter.

More information in the DM: §6 §20 §21 §26 §27

Let's suppose that the player is in room ORIGIN, where he issues a GO command which takes him into a DESTINATION room:

	Library action	Commentary
1	ORIGIN.before();	Then skip to Step 12 (without movement) if **before** property includes `Go: ... rtrue;`
2	location = DESTINATION;	Also move player to DESTINATION
3	location.after();	Then skip to Step 12 if **after** property includes `Go: ... rtrue;`
4	location.initial();	... if location provides this property
5	NewRoom();	... if this optional Entry Point exists
6	print location's room name	Then skip to Step 9 if location has a **visited** attribute and the game isn't in VERBOSE mode
7	location.describe();	... if location provides this property; then skip to Step 9
8	location.description();	
9	list objects in location	According to standard description rules
10	LookRoutine();	... if this optional Entry Point exists
11	give location visited;	Also award ROOM_SCORE points if location has **scored** and **~visited** attributes
12	run daemons and timers	
13	location.each_turn();	... if location provides this property
14	print command prompt ">"	

> **Notes**: The **initial** property can be either a routine -- which **location.initial()** runs -- or a string -- which **location.initial()** prints; the same applies to the **description** and **each_turn** properties. Also, this list excludes the rarely-used Entry Points **GamePreRoutine()** (called before Step 1) and **GamePostRoutine()** (called before Step 4 *and* before Step 12) -- processing then skips to Step 12 if any of these returns **true**.

Trapping "Go" actions is not limited to prevent PC movement or printing messages; it may perform as much mischief as you desire. You could for instance silently detect arrival into a room so that a daemon or a timer is triggered unbeknownst to the player:

```
Object  cell "Cell"
  with  description "A small cube of solid stone walls.",
        before [; Go: StopTimer(self); ],
        after  [; Go: StartTimer(self, 4); ],
        time_left 0,
        time_out [;
            self.w_to = "The door is jammed.";
            give self ~light;
            "The door suddenly slams shut.";
            ],
        w_to torture_chamber,
  has   light;
```

Finally, here are two Room classes which print a message as you move from one room to another. The first reacts in the old room, before the move happens, while the second waits until you reach the new room:

```
Class Room
 with description "UNDER CONSTRUCTION",
      before [ dirProp;
        Go:              ! 'noun' contains a compass object (eg e_obj);
          dirProp = noun.door_dir;   ! convert it to direction property
                                     ! (eg e_to)
          if (self provides dirProp && self.dirProp ofclass Object &&
             (self.dirProp hasnt door || self.dirProp has open))
              print "You leave the room.^";
      ],
  has light;

Class Room
 with description "UNDER CONSTRUCTION",
      after [;
        Go: print "You leave the room.^";
            <<Look>>;
      ],
  has light;
```

Where have I been?

The **location** and **real_location** variables specify the player's current room. You may wish to know the room you were in previously; this isn't held in any library variable, so you need to find a way of remembering it. Here's one method.

```
Object roomStack
  with stack 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0,
       push [ room i;
           for (i=self.#stack/2-1 : i>0 : i--)
               self.&stack-->i = self.&stack-->(i-1);
           self.&stack-->i = room;
       ],
       pop [ room i;
           room = self.&stack-->0;
           for (i=0 : i<self.#stack/2-1 : i++)
               self.&stack-->i = self.&stack-->(i+1);
           self.&stack-->i = nothing;
           return room;
       ],
       peek [ i;
            if (i<0 || i>=self.#stack/2) return nothing;
            return self.&stack-->i;
       ];

Class   Room
  with  description "UNDER CONSTRUCTION",
        after [;
          Go: roomStack.push(real_location);
        ],
  has   light;
```

This is slightly more complex than is strictly necessary, but it makes for an interesting example. We create a **roomStack** object, whose role is to provide both storage for the 32 most recently-visited locations (in its **stack** property), and also three property routines for manipulating the stack. **roomStack.push**(*room*) pushes *room* onto the stack, **roomStack.pop**() pops the top *room* off the stack, and **roomStack.peek**(*i*) returns the *i*th visited room: 0 is the current location, 1 is the previous room, 2 is the one before that, and so on. In this example we don't actually need the **pop** property: it's defined only for completeness.

As in the previous topic, we create a Room class with an **after** property; its role here is to push the current location onto the stack. Then, to print the name of the room you were in previously, you might type:

```
print "You've just come from ", (name) roomStack.peek(1), ".^";
```

While we're on the subject, here's an associated technique: enabling the player to jump directly to a previously-visited room. To do this, you need to give each of your rooms a sensible **name** property, and then add this code:

```
[ GoRoomSub;
    if (noun == location) "But you're here already!";
    print "If only it was always this easy to go to the...^";
    PlayerTo(noun);
];


[ scope_room x;
    switch (scope_stage) {
     1: rfalse;
     2: objectloop (x ofclass Room)
            if (x has visited) PlaceInScope(x);
        rtrue;
     3: "You can't quite remember how to get there.";
    }
];


Extend 'go' first
    * 'to' scope=scope_room     -> GoRoom;
```

How can I parse a number?

You might need to deal with numeric input in a command like ADJUST DIAL TO 3 or SET COURSE TO 180. The Library file `Grammar.h` includes some suitable grammar:

```
  Verb 'set' 'adjust'
      * noun              -> Set
      * noun 'to' special -> SetTo;
```

That **special** token must be doing the business, but trying to find it in the DM4 proves difficult -- it's not described anywhere! Going back to the DM3 gives us the ominous warning "any single word or number (obsolete and best avoided)" -- hardly an auspicious start -- but it's part of the library and it *does* work. So here's an abstract scenery object which can be SET:

```
 Object  course "course"
   with  name 'course' 'heading' 'bearing' 'direction',
         description [; "Your current course is ", self.number, " degrees.";],
         number 0,               ! current course in degrees 0..359
         before [;
             SetTo: switch (second) {
                      0 to 359: self.number = second;
                      360:      self.number = 0;
                      default:  "There are only 360 degrees in a circle!";
                    }
                    "You set the course to ", self.number, " degrees.";
             Examine: ;
             default: "Don't be silly!";
         ],
         found_in [; return true; ],
   has   scenery;
```

The object's **before** property traps the **SetTo** action, and since **special** was the second token in the grammar line, the numeric value is available in the **second** variable. SET and EXAMINE are the only supported verbs: any other action results in "Don't be silly!". The object uses **found_in** to make itself present throughout the game; you could instead simply position it in one room like the ship's bridge, or restrict it to a single area:

```
          found_in [; return (location ofclass OnBoardShip); ],
```

Employing an obsolete token doesn't give a warm fuzzy feeling, so let's update the grammar to use the (properly documented) **number** token instead. While we're here, we'll also extend it to handle SET COURSE TO WEST, and various shortened forms and synonyms:

More information in the DM: §31

```
  Extend 'set' replace
      * noun                              -> Set
      * number                            -> Set
      * 'to'/'towards'/'for' noun         -> SetTo
      * 'to'/'towards'/'for' number       -> SetTo
      * noun 'to'/'towards'/'for' noun    -> SetTo
      * noun 'to'/'towards'/'for' number  -> SetTo;

  Verb 'steer' 'sail' 'navigate' 'head' 'aim' = 'set';
```

Here's the revised 'course' object to deal with this lot; it's less complex than it looks:

```
Object  course "course"
  with  name 'course' 'heading' 'bearing' 'direction',
        description [; "Your current course is ", self.number, " degrees."; ],
        number 0,               ! current course in degrees 0..359
        try_number [ dir;       ! dir is a number of degrees?
            switch (dir) {
              0 to 359: ;
              360:      dir = 0;
              default:  dir = -1000;
                        print "There are only 360 degrees in a circle!^";
            }
            return dir;
        ],
        try_object [ dir;       ! dir is a direction object?
            switch (dir) {
              n_obj:    dir = 0;
              e_obj:    dir = 90;
              s_obj:    dir = 180;
              w_obj:    dir = 270;
              default:  dir = -1000;
                        print "That's not a proper direction!^";
            }
            return dir;
        ],
        try_dict [ dir;     ! dir is a dictionary word?
            switch (dir) {
              'north','n//':    dir = 0;
              'east','e//':     dir = 90;
              'south','s//':    dir = 180;
              'west','w//':     dir = 270;
              default:          dir = -1000;
                                print "That's not a proper direction!^";
            }
            return dir;
        ],
        react_before [ x;
            Set,SetTo:
                if (second == nothing)
                    if (inp1 == 1) x = self.try_number(noun);
                    else           x = self.try_object(noun);
                else
                    if (inp2 == 1) x = self.try_number(second);
                    else           x = self.try_object(second);
                if (x < 0) rtrue;
                self.number = x;
                "You set the course to ", self.number, " degrees.";
        ],


        before [;
            Examine: ;
            default: "Don't be silly!";
        ],
        found_in [; return true; ],
  has   scenery;
```

We've added three property routines -- **try_number**, **try_object**, and **try_dict** -- to provide validation and data mapping. We use these routines when we trap the Set and SetTo actions, which we do in a **react_before** property rather than in **before**: this enables us to handle command like STEER 180 and SAIL TO THE SOUTH in which the 'course' object isn't actually mentioned. Notice the use of the library variables **inp1** and **inp2**, normally the same as **noun** and **second** respectively, but set to 1 when **noun** or **second** contains a literal number rather than an object reference.

More information in the DM: §6

Just to round things off, let's add an NPC who can do the steering on our command; that is, SAY SOUTH TO SAILOR or SAILOR,SET COURSE TO WEST or SAILOR,STEER 180 or even SAILOR,270.

```
  Object  "sailor"
    with  name 'sailor' 'pilot' 'helm' 'helmsman' 'steersman',
          life [ x;
              Answer:
                  if (consult_words ~= 1) return L__M(##Answer,1,noun);
                  x = TryNumber(consult_from);
                  if  (x ~= -1000) x = course.try_number(x);
                  else             x = course.try_dict(noun);
                  if (x < 0) rtrue;
                  course.number = x;
                  "~Course set to ", course.number, " degrees, sir.~";
          ],
          orders [ x;
              Go,Set,SetTo:
                  if (second == nothing)
                      if (inp1 == 1) x = course.try_number(noun);
                      else           x = course.try_object(noun);
                  else
                      if (inp2 == 1) x = course.try_number(second);
                      else           x = course.try_object(second);
                  if (x < 0) rtrue;
                  course.number = x;
                  "~Course set to ", course.number, " degrees, sir.~";
              NotUnderstood:
                  x = course.try_number(special_number);
                  if (x < 0) rtrue;
                  course.number = x;
                  "~Course set to ", course.number, " degrees, sir.~";
          ],
    has   animate;
```

In this skimpy implementation, the **life** property handles only Answer (SAY *word* to SAILOR, where *word* is a single number or a compass direction in the dictionary). The **orders** property deals with Go (SAILOR,EAST and SAILOR,GO EAST), Set (SAILOR,SET 90 and SAILOR,SAIL EAST), and NotUnderstood (SAILOR,90 which uses the **special_number** library variable).

More information in the DM: §16 §17 §18

Which action is triggered by each verb?

To deal with a particular english verb in a property like **before** or **life**, you need to know which **action** is triggered, because it's actions that you intercept, not verbs. Usually, you can do this by finding the verb in `Grammar.h`; for example, as we showed in the previous topic, you'll discover that the SET verb triggers the Set and SetTo actions.

Sometimes, one action gets translated to another during processing, which can be a bit confusing. For example, Drop (a worn object) can become Disrobe, Examine (a container object) can become Search, and Transfer can variously become PutOn or Drop or Insert. Here are two techniques to help you discover what's going on.

First, use the ACTIONS debugging command, which tells you the current action and its parameters:

```
  >ACTIONS
  [Action listing on.]

  >SIT ON ROCK
  [ Action Enter with noun 29 (rock) ]
  That's not something you can sit down on.
```

More information in the DM: §7

Second, where that doesn't work (for example, sometimes when issuing an order to an NPC), include this line of debugging code at the start of your **before** or **life** or **orders** property:

```
  orders [ x;
      #ifdef DEBUG; print "[Action=", (debugAction) action, "]^"; #endif;
      Go,Set,SetTo:
      ...
```

which tells you this:

```
  >SAILOR,EAST
  [Action=Go]
  "Course set to 90 degrees, sir."

  >SAILOR,100
  [Action=<fake action 9>]
  "Course set to 100 degrees, sir."
```

The fake actions are defined in `Parser.h`; 0 is Receive, through to 9 is NotUnderstood.

Can I distinguish SIT ON BED from LIE ON BED?

Inform's standard grammar maps several verbs -- including ENTER, GET INTO/ONTO, STAND ON, SIT ON and LIE ON -- to the single action of Enter. You can Enter an object which has an **enterable** attribute, so a basic bed object looks and works something like this:

More information in the DM: §12 §15

```
  Object  -> bed "bed"
    with  name 'bed',
          description "It's a pretty regular bed.",
    has   enterable supporter static;

  Your bedroom
  A spartan cell.

  You can see a bed here.

  >SIT ON BED
  You get onto the bed.

  >GET OFF BED
  You get off the bed.

  Your bedroom
  A spartan cell.

  You can see a bed here.
```

There are a couple of potential problems with this. First, LIE, SIT and STAND are treated identically -- you're either on the bed or you're not -- and second the act of getting off the bed causes the room description to be replayed. This is because it's handled by the Exit action, which is perhaps geared more towards climbing out of a **container** such as a vehicle or a wardrobe than simply getting off a **supporter** such as a bed or chair.

In order to deal with these issues, we'll build a Bedlike class, useful for beds, sofas, benches and similar feathered horizontal surfaces. In fact, we'll do it twice, once using the methods described above to Extend the standard grammar, and then again using a different technique which keeps the grammar unchanged. Here's the extended grammar for the first solution:

```
  [ SitOnSub;   <<Enter noun>>; ];
  [ LieOnSub;   <<Enter noun>>; ];
  [ StandOnSub; <<Enter noun>>; ];

  Extend only 'sit' replace
      * 'on' 'top' 'of' noun      -> SitOn
      * 'on'/'in'/'inside' noun   -> SitOn;
```

```
Extend 'lie' replace
    * 'on' 'top' 'of' noun      -> LieOn
    * 'on'/'in'/'inside' noun   -> LieOn;

Extend 'stand' replace
    *                           -> Exit
    * 'up'                      -> Exit
    * 'on' 'top' 'of' noun      -> StandOn
    * 'on' noun                 -> StandOn;

Extend only 'climb'
    * 'on' 'top' 'of' noun      -> StandOn
    * 'on' noun                 -> StandOn;
```

You'll see that we're pulling apart the existing grammar -- in which all of those verbs trigger the Enter action -- and redirecting them to distinct SitOn, LieOn and StandOn actions. Then, to avoid breaking any other objects which rely on the standard behaviour, we create default handlers for the new actions which simply invoke Enter, so that the whole thing works exactly as before. However... we can now create a Bedlike class which *doesn't* conform to the normal pattern, but instead intercepts the new actions:

```
Class   Bedlike
  with  sit_or_lie false,
        before [;
          Enter,StandOn:
            if (self == parent(player) && self.sit_or_lie == false)
                return L__M(##Enter,1,self);
            move player to self;
            self.sit_or_lie = false;
            if (AfterRoutines() || keep_silent) rtrue;
            "You stand on ", (the) self, ".";
          SitOn:
            if (self == parent(player) && self.sit_or_lie == action)
                return L__M(##Enter,1,self);
            move player to self;
            self.sit_or_lie = action;
            if (AfterRoutines() || keep_silent) rtrue;
            "You sit on ", (the) self, ".";
          LieOn:
            if (self == parent(player) && self.sit_or_lie == action)
                return L__M(##Enter,1,self);
            move player to self;
            self.sit_or_lie = action;
            if (AfterRoutines() || keep_silent) rtrue;
            "You lie on ", (the) self, ".";
        ],
        react_before [;
          Exit:
            if (self == parent(player)) {
                move player to parent(self);
                if (self.sit_or_lie) {
                    self.sit_or_lie = false;
                    if (AfterRoutines() || keep_silent) rtrue;
                    "You stand up.";
                }
                if (AfterRoutines() || keep_silent) rtrue;
                "You climb off ", (the) self, ".";
            }
        ],
  has   supporter static;

Bedlike -> bed "bed"
  with  name 'bed',
        description "It's a pretty regular bed.",
        after [; LieOn: "You settle down for a short nap."; ];
```

The class's **before** property deals with the regular Enter action, and with our new StandOn, SitOn and LieOn actions. In each case, the action first tests if it has anything useful to do and if not -- for example SIT ON BED when the player is already seated -- displays the standard refusal message "But you're already on the bed". Otherwise it moves the player into position and uses the **sit_or_lie** property to remember whether he's sitting or lying down. Next, it calls the **AfterRoutines()** library routine in case the Bedlike object has provided an **after** property; a **true** return value from such a property would prevent us displaying our confirmation message, as would a **true** value in the **keep_silent** library variable.

Because the Exit action isn't directed to a specific object, we can't handle it using a **before** property; instead, we **react_before** an Exit and trap the action only if this Bedlike object is the player's parent. Effectively, we've completely replaced the Library's standard processing for Enter and Exit of Bedlike objects, and so we've now no need for an **enterable** attribute. And here's the outcome:

```
Your bedroom
A spartan cell.

You can see a bed here.

>GET ON BED
You stand on the bed.

>GET OFF BED
You climb off the bed.

>SIT ON BED
You sit on the bed.

>LIE ON BED
You settle down for a short nap.

>GET OFF BED
You stand up.
```

In our alternative approach, we don't need to Extend the grammar; the class's **before** property deals only with the Enter action, and distinguishes SIT and LIE by inspecting the Library variable **verb_word** which holds the dictionary value for the verb in the current command. The remainder of the class definition is the same as our previous example, and it works identically:

More information in the DM: §18

```
Class   Bedlike
  with  sit_or_lie false,
        before [;
          Enter:
            if (verb_word == 'sit' or 'lie') {
                if (self == parent(player) && self.sit_or_lie == verb_word)
                    return L__M(##Enter,1,self);
                move player to self;
                self.sit_or_lie = verb_word;
                if (AfterRoutines() || keep_silent) rtrue;
                if (verb_word == 'sit')
                    "You sit on ", (the) self, ".";
                else
                    "You lie on ", (the) self, ".";
            }
            else {
                if (self == parent(player) && self.sit_or_lie == false)
                    return L__M(##Enter,1,self);
                move player to self;
                self.sit_or_lie = false;
                if (AfterRoutines() || keep_silent) rtrue;
                "You stand on ", (the) self, ".";
            }
        ],
        ...
```

Incidentally, should you wish to distinguish between, for example, LIE IN BED and LIE ON BED, you could either Extend the grammar further, by defining separate LieIn and LieOn actions, or you could use this routine to return the word following the verb, and then test it against 'in' and 'on':

```
[ WordAfterVerb w;
    w = verb_wordnum + 1;
    if (w > parse->1) return 0; ! Nothing following the verb
    w = parse-->(w*2 - 1);
    if (w) return w;            ! Following word has this value
    return -1;                  ! Following word not in dictionary
];
```

Which verb did the player use?

When the player types something like OFFER WATER TO HORSE, the parser sets up four variables to represent the command: **action** (in this example, 'Give') defines what's to be done, along with **noun** (the 'water' object) and **second** (the 'horse' object); in addition, **actor** (usually 'selfobj') is the object to whom the command is directed.

More information in the DM: §16 §18 §30

It's easy to process the **noun** and **second** objects, including if necessary printing their names: print (name) noun; is all that it takes. It's a bit trickier, though, to get back to the original command verb if that's what you want to display. You've got three sources of information:

- the **action** variable which we've just mentioned; however, this hold the action triggered by the verb rather than the verb itself (for example, the verbs GIVE, FEED, OFFER and PAY all cause the same 'Give' action);
- the **verb_word** variable, which hold the actual verb's dictionary address; however, this value reflects only the verb itself, not any subsequent preposition which may modify its effect (for example, LOOK, LOOK AT, LOOK IN and LOOK UNDER each triggers a different action, but verb_word will contain 'look' in all cases);
- the **buffer** and **parse** arrays, which hold the actual characters typed by the player.

There are a few further complications; the verb may be:

- *abbreviated* -- for example, L is treated as equivalent to LOOK;
- *truncated* -- for example, PHOTOGRAPH is stored in the dictionary as 'photograp';
- *implied* -- for example, NORTH is taken to mean GO NORTH;
- *not the first word* -- as in, for example, HORSE,EAT HAY.

What all this means is that you've got quite a bit of work to do if you want to be able to reproduce the original command as the player mentally formulated it. Here's one way of tackling the problem:

```
[ PrintVerbWord
    i j k;

    ! print the (possibly implied, possibly abbreviated) verb

    if (action == ##Go && verb_word ~= 'go' or 'run' or 'walk' or 'leave')
        print "go";
    else
        if (LanguageVerb(verb_word) == false) { ! expand an abbreviation?
            #Ifdef TARGET_ZCODE;
            j = parse->(4*verb_wordnum + 1);    ! start posn in buffer
            k = parse->(4*verb_wordnum);        ! number of characters
            #Ifnot; ! TARGET_GLULX
            j = parse-->(3*verb_wordnum);       ! start posn in buffer
            k = parse-->(3*verb_wordnum - 1);   ! number of characters
            #Endif; ! TARGET_
            for (i=0 : i<k : i++) print (char) Lowercase(buffer->(i+j));
        }
    ! possibly append a modifying preposition

    switch (action) {
      ##Ask:
        print " about";
      ##AskFor:
        print " for";
      ##AskTo:
        print " to";
      ##Consult:
        if (verb_word == 'look' or 'l//') print " up";
        if (verb_word == 'read') print " about";
      ##Disrobe:
        if (verb_word == 'take') print " off";
      ##Drop:
        if (verb_word == 'put') print " down";
      ##EmptyT:
        if (verb_word == 'empty') print " into";
      ##Enter:
        if (verb_word == 'get' or 'go') print " in";
        if (verb_word == 'lie' or 'sit' or 'stand') print " on";
      ##Examine:
        if (verb_word == 'look' or 'l//') print " at";
      ##Exit:
        if (verb_word == 'get') print " off";
        if (verb_word == 'stand') print " up";
      ##GetOff:
        if (verb_word == 'get') print " off";
      ##Go:
        print " "; LanguageDirection(noun.door_dir);
      ##Insert:
        if (verb_word == 'drop' or 'discard' or 'put' or 'throw')
            print " in";
      ##JumpOver:
        print " over";
      ##LookUnder:
        if (verb_word == 'look' or 'l//') print " under";
      ##PutOn:
        if (verb_word == 'drop' or 'discard' or 'put' or 'throw')
            print " on";
      ##Remove:
        if (verb_word == 'get' or 'take') print " off";
      ##Search:
        if (verb_word == 'look' or 'l//') print " in";
      ##SetTo:
        print " to";
      ##SwitchOff:
        print " off";
      ##SwitchOn:
        print " on";
      ##Take:
        if (verb_word == 'pick') print " up";
      ##Tell:
        if (verb_word == 'tell') print " about";
      ##ThrowAt:
        print " at";
      ##Wear:
        if (verb_word == 'put') print " on";
    }
];
```

How do 'meta' verbs work?

As we said earlier, a meta verb controls the game itself, rather than affecting the model world within the game. When the parser finds that the player has typed a verb tagged as 'meta', it sets a variable -- also called **meta** -- to true. This has two main effects:

1. prior to performing the requested action, the **BeforeRoutines()** processing is omitted -- that is, **GamePreRoutine()** isn't called, and no **react_before** or **before** properties are executed.
2. after performing the requested action, the entire **InformLibrary.end_turn_sequence()** processing is omitted -- that is, the time and turns counters are unchanged, no daemons or timers run, and no **each_turn** properties are executed.

Also, the meta verbs are of Group 1; this, like Group 3, doesn't change the model world, there isn't an 'afterwards' that differs from the previous state, and so the **AfterRoutines()** processing is omitted -- that is, no **react_after** or **after** properties are executed, and **GamePostRoutine()** isn't called.

Occasionally, you may find the need to set **meta** to true in a verb's action routine. Since this is called *between* steps 1 and 2 above, the result is to prevent the **end_turn_sequence()** processing, but not any **BeforeRoutines()** processing (because by then it's too late).
Here's a neat technique allowing a verb to be both meta and non-meta. The author wants to define ABOUT FACE and ABOUT TURN verbs, to cause the PC to look behind him. This is probably a Group 2 action, since it'll change the model world slightly. However, he also wants to permit ABOUT as a synomym for HELP and INFO -- a meta verb providing information about the game itself. This first attempt won't work, because the same verb can't appear in more than one grammar:

```
  [ HelpSub; ... ];
  [ AboutFaceSub; ... ];

  Verb meta 'about' 'help' 'info'
      *                -> Help;

  Verb 'about'
      * 'face'/'turn' -> AboutFace;
```

More information in the DM: §31

But this second try works fine:

```
  Verb meta 'help' 'info'
      *                -> Help;

  [ isMeta; meta = true; return GPR_PREPOSITION; ];

  Verb 'about'
      * 'face'/'turn' -> AboutFace
      * isMeta         -> Help;
```

isMeta() is an example of a "general parsing routine". In this case, it's pretending to have matched a preposition (like 'face' or 'turn' in the line above), though actually it's done nothing of the sort. All that's happened is that **meta** has been set to true; the effect is that ABOUT with nothing following it is now handled the same as HELP and INFO.

7 · Bothered By Bugs

What can I expect when I run my first program?

Once you've fixed all the annoying little syntax errors which previously stopped your game from compiling (see What can I expect when I try to compile my program?), you win the pleasure of actually being able to play it. All too often that pleasure is short-lived; almost immediately you'll start to notice deficiencies -- some subtle, some glaringly obvious -- that need to be improved or mended. This is a perfectly normal experience, and you should expect to spend a considerable period exploring the ways in which your game fails to behave as you'd intended.

When correcting the problems that you'll inevitably discover, you should move slowly and cautiously. Work on one issue at a time: identify what's wrong, change the program (maybe only a single line), recompile, and test what you've just done. If you're lucky, you've fixed the problem. If you're unlucky, not only may the problem still remain, but something else may have gone wrong. This is also all too common, but by recompiling frequently and immediately testing the change, you'll much more quickly home in on what's gone wrong and how to fix it. One of the biggest causes of frustration comes from making a large number of changes all at once, not recompiling until every modification is in place, then finding that the game doesn't work nearly as well as it did before... and not having any idea which of the many changes did the real damage.

By the way, if you're about to embark on major surgery to a working program, *save a copy of the file* before proceeding. That way, if things go seriously pear-shaped, you can always revert to that copy, rather than trying to undo all the individual changes.

Help! What's wrong with my code?

More information in the DM: §40

Just because you've found all the mistakes which prevented your game from compiling, and you've resolved the warning messages -- you *have* got rid of them, haven't you? -- it doesn't follow that your game is free from bugs. Here are some common traps for the unwary Inform novice.

1. The **"*string*";** statement prints the string, outputs a newline, *and then returns*. This is probably the most common Inform gotcha. If you write these lines:

```
"The fire is ";
if (self has general) print "blazing fiercely";
else print "smouldering gently";
print " in the grate.";
```

then the compiler will complain that the second line is unreachable. It *won't* complain if you write this, though the situation is just the same:

```
if (self has general) "The fire is blazing fiercely";
else "The fire is smouldering gently";
print " in the grate.";
```

Fix: change to ordinary **print** statements:

```
if (self has general) print "The fire is blazing fiercely";
else print "The fire is smouldering gently";
print " in the grate.";
```

2. An **if** and an **else** each control the single statement which follows. If you write these lines then you're like to see that the fire "continues to blazesuddenly flares up".

```
if (self has general) print "continues to blaze"
```

```
else give self general; print "suddenly flares up";
```

Fix: wrap braces around the multiple statements:

```
if (self has general) print "continues to blaze"
else { give self general; print "suddenly flares up"; }
```

3. An **else** relates to the immediately-preceding **if**. If you write these lines then you're likely to see just "The grate is".

```
print "The grate is ";
if (grate has on)
    if (self has general) print "hot";
else print "cold.";
```

Fix: wrap braces around the second **if**, or combine the two **if**s:

```
print "The grate is ";
if (grate has on) {
   if (self has general) print "hot";
   }
else print "cold.";
print "The grate is ";
if (grate has on && self has general) print "hot";
else print "cold.";
```

4. Don't confuse logical not '**~~**' with bitwise not '**~**'. If you write these statements, then you'll *always* print "ok":

```
if (~self in location) print "ok";
```

This is because the condition **self in location** evaluates to either false (0) or true (1). A bitwise 'not' toggles each individual bit: '~' applied to 0 ($$0000000000000000) gives -1 ($$1111111111111111), which is true; '~' applied to 1 ($$0000000000000001) gives -2 ($$1111111111111110), which is *also* true.

Fix: use '~~' instead:

```
if (~~self in location) print "ok";
```

A logical 'not' toggles the true/false meaning of all the bits taken together; '~~' applied to 0 ($$0000000000000000) gives 1 ($$0000000000000001), which is true; '~~' applied to 1 ($$0000000000000001) gives 0 ($$0000000000000000), which is false.
For the same reason, don't confuse '&&' and '||' (which are quite commonly used) with '&' and '|' (found much more rarely).

5. Don't apply the test '== true' to something which can have a value other than 0 or 1. If you write these statements, then you'll *not* print "ok" if number is 2,3,...:

```
if (self.number == true) print "ok";
```

Fix: change the test to one of these:

```
if (self.number) print "ok";
if (self.number ~= false) print "ok";
```

6. Don't write just **number** (or any other property name) instead of **self.number** (or *object***.number**) -- things will mysteriously fail to work.

7. Beware of creating **name** properties with impossible values, such as single letters 'g' (that's a character constant, not a dictionary word) and phrases 'g string' (the parser tests only single words, not phrases with spaces in them). This is correct:

```
        name 'g//' 'string' 'gstring' 'g-string',
```

8. If you refer to a routine *without* '()' afterwards, the result is the address of that routine rather than the value which it returns:

```
    x = myRoutine;    ! wrong: x contains the address of myRoutine
    x = myRoutine();  ! correct: x contains the return value from running
                      ! myRoutine
```

9. If you get unexpected 0s and 1s appearing, check the next topic.

10. Arithmetic is limited to numbers in the range -32768..0..32767. If you add or multiply two numbers giving a result outside that range, the outcome may be unexpected.

11. Don't use the **found_in** property for moveable objects; they'll magically return home if the player tries to walk off with them. **found_in** is intended only for objects with a **scenery** attribute.

Why do I get spurious 0s and 1s in my printout?

Almost certainly, because you've called a routine in the middle of the **print** statement. The next answer describes the syntax trick to avoid this problem, but here's why it happens. Consider this tiny fragment of code:

```
  [ Squared x; print x*x; ];

  print "The square of ", 4, " is ", Squared(4), ".^";
```

What happens here is: the **print** statement outputs "The square of 4 is ", then calls the routine. The routine outputs the answer of 16 and returns. The **print** statement then outputs 1 -- the value returned by the routine -- followed by the fullstop and newline. The value returned by the routine? Yes, *every* routine returns a value, which depends on the last executed statement in the routine.

If the last executed statement is...	**then the value returned by the routine is...**
return *value*;	*value*

return; or
rtrue;
	true (1)

rfalse;	**false** (0)
] at the end of a standalone routine	**true** (1)
] at the end of an embedded routine	**false** (0)

More information in the DM: §1.7

As an aide-memoire, remember the letter pairs **ST...EF**:
Standalone routines end as **T**rue.
Embedded routines end as **F**alse,

A 'standalone' routine is one that has a name and exists in its own right; like Squared above. An 'embedded' routine is one without a name which you include within an object definition as the value of a property:

More information in the DM: §3.5

```
    with  description [; code to print an object's description ],
          ...
```

What's the difference between Squared(x) and (Squared) x?

It's the difference between the spurious 0 and 1 problem occurring, and not. If your program reads:

```
  print "The square of ", 4, " is ", Squared(4), ".^";
```

then the output you'll get is:

```
  The square of 4 is 161.
```

However, if you modify the program, leaving the routine unchanged but treating it as a print rule, so that the line now reads:

```
  print "The square of ", 4, " is ", (Squared) 4, ".^";
```

then the output you'll get, correctly, is:

```
  The square of 4 is 16.
```

This funny little syntax trick, which applies only when printing, has the effect of calling the specified routine but not outputting the value which it returns. That is, if the routine itself includes a **print** statement, that's what gets output; should the routine happen *not* to include a **print** statement, nothing gets output.

More information in the DM: §1.12

How can I make the debugging process easier?

You're going to be doing a lot of it; much IF authorship comprises brief periods of enjoyable creativity separated by longer spells of laborious testing and frustrating bug hunts. So, if only from a sense of self-preservation, you'd be well advised to get to grips with Inform's debugging capabilities. Two techniques are of particular value:

Automated replay. During testing, you'll be going over the same ground time and time again. You can save yourself from sore fingers, and also catch those nasty regression problems where an innocuous change here has an unexpected impact on well-tested code there, by setting up a script of commands which you can REPLAY over and over. It's easy to create such a script with RECORDING ON, typing in the commands, and then closing with RECORDING OFF. Since the script is just an ASCII file, one command per line, you can also create it using an editor, and similarly you can extend an existing script by including additional commands in appropriate places.

More information in the DM: §7.1

Object inspection and manipulation. The extra debugging verbs provide internal information, and enable you to fiddle with objects' locations; you can save yourself time by cheating in this way, rather than having to modify the source and recompile every time. The most commonly-used commands are these:

More information in the DM: §7.3 §7.5

Use this command...	to do this...
SHOWOBJ	Display the properties and attributes of the current location.
SHOWOBJ *obj_name*	Display the properties and attributes of *obj_name*.
TREE	Display the complete object hierarchy.
TREE *obj_name*	Display the object hierarchy below *obj_name*.
SCOPE	List the objects currently in scope.
PURLOIN *obj_name*	Add *obj_name* to your inventory.
ABSTRACT *obj_name* TO *parent_obj_name*	Make *obj_name* a child of *parent_obj_name*.
GONEAR *obj_name*	Jump to the location containing *obj_name*.
GOTO *obj_number*	Jump to the location numbered *obj_number*.

For more powerful interventions, such as changing variables and setting attributes, you can bring the Infix debugger into play (just include the -X switch when compiling). Infix is slightly harder to master than the basic debug verbs, but can be a godsend when the going gets tough.

More information in the DM: §7.6

If you're already familiar with the UNIX tool **gdb**, you might like to try the Nitfol interpreter, which incorporates a gdb-like debugger. On the other hand, if you're not already a gdb expert, you probably shouldn't try learning it here.

There's some more information about compiling and debugging on Roger Firth's InfLight pages

Why does my game mention "a apple"?

Apparently, the business of outputting "an" rather than "a" before a word beginning with a vowel is by default handled by the interpreter rather than the compiler; unfortunately, some interpreters get it wrong (EXAMPLE?). To be safe, give objects which need it an explicit **article** property:

```
Object  apple "apple"
  with  name 'apple' 'fruit',
        article "an",
        ...
```

What were Vile Zero Errors From Hell?

That was the fanciful name for a nasty error condition which crashed the Z-machine if X happened to be zero when executing statements like these:

For more background information, see Andrew Plotkin's detailed explanation.

```
child(X), parent(X), sibling(X), ...
if (X.property == ...) ..., X.property = ...
if (X has attribute) ..., give X attribute
move X to ..., move ... to X, remove X
```

Since the availability of Strict error checking in Library 6/21, it's rare for games to crash in this way; instead, you'll just get a runtime error message. You still need to fix the error, but at least nowadays you know pretty well where to start looking.

I've found an Inform problem -- what should I do?

Like any other complex system, the Inform software and its associated documentation contain errors of varying severity. If you think that you've found something wrong, don't just sigh and move on: please, report the problem. The first thing to do, obviously, is to check and re-check your suspicions. Once you've convinced yourself that something really is wrong, then you should study the various published lists; it's quite possible that you're not the first person to have happened upon the problem. There are pages devoted to various topics; each tells you who to write to if you wish to report a new problem or add another suggestion:

problems with the compiler
problems with the library
errors in the Designer's Manual
errors in the Beginner's Guide
suggestions for future enhancements

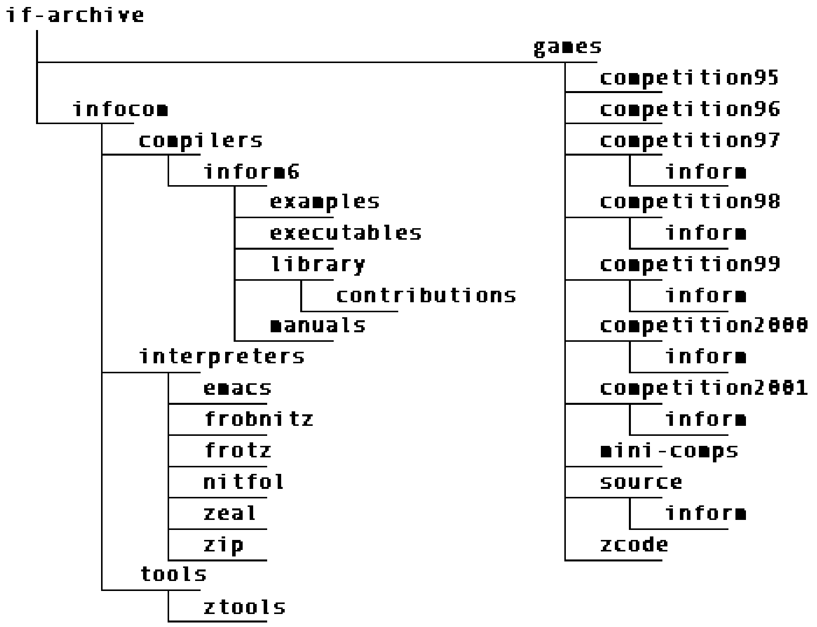
If your problem lies elsewhere -- perhaps in the interpreter you're using, or in a library contribution -- then you should write directly to its author. It's generally considered impolite to baldly announce in the rec.arts.int-fiction newsgroup that you've found a bug, though you *can* ask for assistance there in determining the exact nature of the problem you're experiencing.

8 · History And Hereafter

Where's this Archive that's mentioned so often?

A truly invaluable service to the whole IF community -- not just us Inform programmers -- is performed by David Kinder and Stephen Granade, the custodians of the IF Archive (and for nine years previously, by Volker Blasius at the GMD in Germany: thank you, thank you, Volker). The Archive is a hierarchy of logically-organized storage locations containing files -- games, solutions, hints, maps, compilers, interpreters, editors, and much more -- which have been created by IF enthusiasts over the past ten or so years. Whatever you're looking for, if it's non-commercial IF, then it's probably in the Archive.

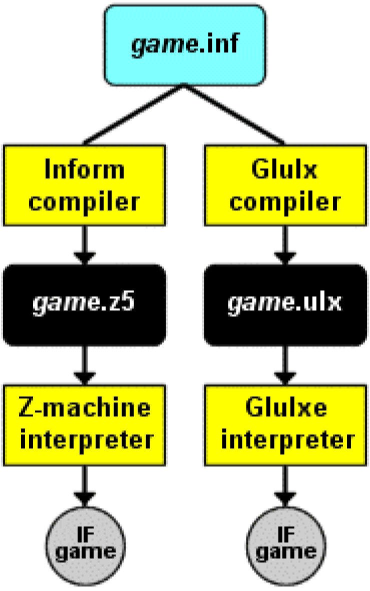
The Archive is, since August 2001, held on a FTP site in America, with mirror sites in other locations around the world. You can inspect its contents by visiting ftp://ftp.ifarchive.org/if-archive/, but you'll probably find it more approachable using the HTTP interface at http://www.ifarchive.org/, since this conveniently gives a description of each file. Because of its scale, the Archive can be a bit daunting at first; here are the locations of most immediate interest to Inform programmers and players.



As you become more assured in your use of Inform, you may create something -- a game, say, or a Library package -- which belongs in the Archive. To upload your file, first get hold of a decent FTP program (for the PC I recommend WS FTP LE which is free for non-commercial use), and then follow these steps:

1. Start your FTP program and connect to host name "ftp.ifarchive.org". Use "Automatic detect" for the host type, "anonymous" as the User ID, and your email address as the password.
2. When you're connected, select the host's `incoming` directory.
3. Normally, select ASCII as the file type; use Binary if you're uploading a `.Z5` or `.Z8` game, or some other non-text file.
4. Copy the file into the Archive's folder, and then close the connection.
5. Send an email message to `submissions@ifarchive.org`, containing three items:
   - the name of the file that you've just uploaded;
   - the location within the Archive where you'd like it to be stored (for example `games/zcode` or `infocom/compilers/inform6/library/contributions`);
   - a short description of the file -- look at similar Archive files to get a feel for what to say and how to say it.

Glulx? What's that all about?



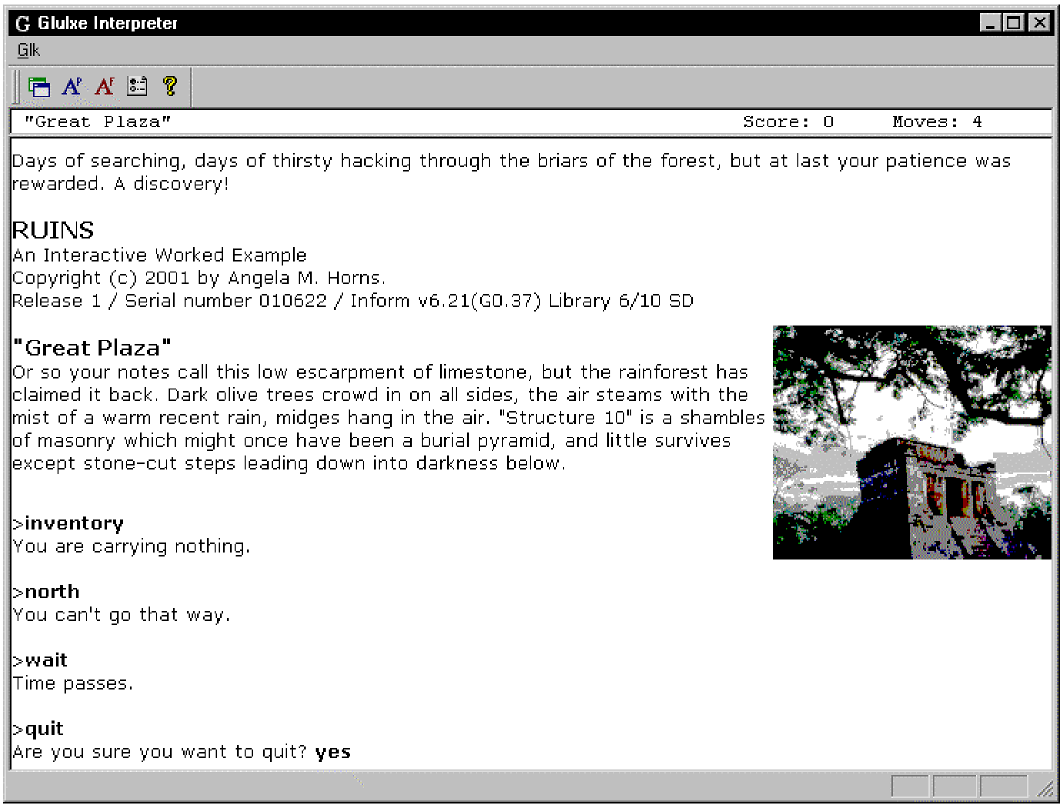
(This is a very brief overview: For a readable and more detailed introduction to this topic, I highly recommend Adam Cadre's Gull pages.)

When you run an Inform interpreter like Frotz or Zip, it's actually implementing, in software, an imaginary computer called the Z-machine. As far as we know, a Z-machine has never existed as a physical pile of components bolted together, but lots and lots of them have been conjured into being using emulation by computer programs.

The Z-machine was devised by the founders of Infocom back in 1979, and it has survived the last twenty-some years pretty well. Nevertheless, its design is nowadays acknowledged to be rather old-fashioned, being short of addressable memory and lacking support for today's sound and graphic standards. In order that Inform games can become larger, louder, and more colourful, the Z-machine needs to be superseded by a reworked Virtual Machine.

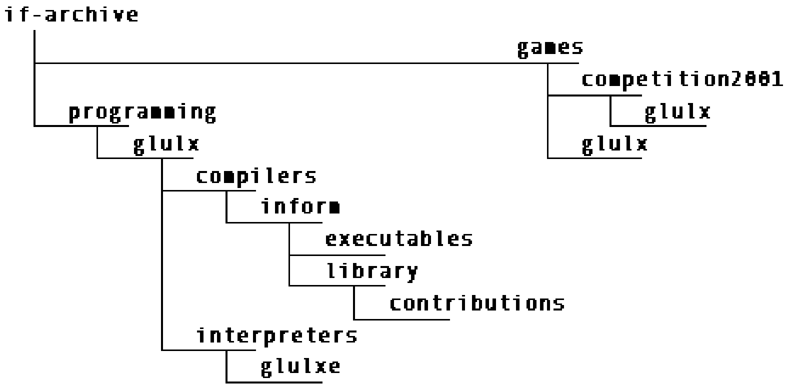
Glulx is that new VM, designed to overcome the Z-machine's limitations; it's the work of Andrew Plotkin (commonly known as Zarf and often to be found around the IF newsgroups). The internal architecture of Glulx is quite different from that of the Z-machine (for example, it's based on computer words of 32 bits, not the 16 bits of the Z-machine) but that isn't too important. What does matter is that Zarf has enhanced the Inform compiler so that it knows about Glulx. This is great news! It means that you don't need to learn anything much that's new; you can just keep on writing Inform games according to the *Designer's Manual*, and the Glulx compiler will turn them into code that runs on Glulxe, the interpreter for the Glulx system.

Remember back when explaining What is Inform?, we illustrated the phrase the "explored ... with an interpreter program" with a screenshot? Well, here's the same game running under Glulx... except that by adding one extra statement we can illustrate our location:



```
 "Great Plaza"                                         Score: 0       Moves: 4

Days of searching, days of thirsty hacking through the briars of the forest, but at last your patience was
rewarded. A discovery!

RUINS
An Interactive Worked Example
Copyright (c) 2001 by Angela M. Horns.
Release 1 / Serial number 010622 / Inform v6.21(G0.37) Library 6/10 SD

"Great Plaza"
Or so your notes call this low escarpment of limestone, but the rainforest has
claimed it back. Dark olive trees crowd in on all sides, the air steams with the
mist of a warm recent rain, midges hang in the air. "Structure 10" is a shambles
of masonry which might once have been a burial pyramid, and little survives
except stone-cut steps leading down into darkness below.

>inventory
You are carrying nothing.

>north
You can't go that way.

>wait
Time passes.

>quit
Are you sure you want to quit? yes
```

Here are the Archive locations of most immediate interest to Glulx programmers and players:



For information on moving towards Glulx, see Roger Firth's JustEnoughGlulx pages

When you're ready to try Glulx, you just need to use the **-G** compiler switch. You'll also have to download an interpreter; if you're using the layout that I suggested earlier, a suitable empty folder is already in place for you to download into. Associate the ".ulx" extension with the Glulxe interpreter, and away you go.

Who, or what, is Platypus?

You can visualise the Inform system as having four major components:

- the **game** -- a file of Inform source code: this is the bit that you write, while the remaining components are universal;
- the **compiler** -- a program designed by Graham Nelson to read Inform source files and generate Z-machine code;
- the **interpreter** -- any of a family of programs, created by different people, which run that code and thus let you play the game;
- the **library** -- several Inform source files defining a sophisticated command-line parser and the skeleton environment and behaviour of Graham's model world which your game inhabits.

When we talk about "the library" we almost always mean the nine files provided by Graham and described in the DM4, which enable you to create objects with standard properties and attributes, to call standard routines, and to manipulate standard data structures. However, there's nothing sacred about those nine files: the Inform code which they contain, though long and complex, is freely available and can if desired be changed or even replaced.

Which is what Anson Turner has done to produce Platypus -- an alternative library which you can use instead of Graham's. Platypus offers roughly the same capabilities as the standard library, but in a manner which is not necessarily identical or interchangeable: you need to choose which library to work with. For example, Platypus replaces the twelve directional properties **n_to**, **e_to**, etc with a single **dirs** property, and enhances the standard **before** and **after** properties with a more flexible seven-stage process. Similarly, some attributes disappear (including **lockable** and **scenery**) while others -- including **inside**, **under** and **upon** -- have been added.

You can download Platypus from the Archive. And, you might be interested in seeing Owen Muniz's version of Cloak of Darkness, modified to use Platypus rather than the standard library.

John Wood has published an HTML version of the Platypus documentation

9 · Worldly Woes (advanced)

How can I get rid of those damn walls?

This topic has been superseded by libray 6/11

There are two problems in this area: EXAMINE NORTH replies "You see nothing special about the north wall." even for external locations, and EXAMINE WALL prompts "Which wall do you mean, the north wall, the south wall ... ?". This code, though slightly messy to implement, fixes those problems and, as a bonus, also supports LOOK [TO THE] NORTH using David Cornelson's CompassLook enhancement.

1. Unavoidably, you need to edit English.h. Right at the start of that file, move the line **IFNDEF WITHOUT_DIRECTIONS;** *up* slightly, and its matching **ENDIF;** *down* slightly, so that they enclose the entire definition and use of **CompassDirection** (this will have no effect on the vast majority of games which don't define WITHOUT_DIRECTIONS):

```
IFNDEF WITHOUT_DIRECTIONS;

Class  CompassDirection
  with article "the", number 0
  has  scenery;

Object Compass "compass" has concealed;

CompassDirection -> n_obj "north wall"
                    with name 'n//' 'north' 'wall',
                    door_dir n_to;
...
CompassDirection -> in_obj "inside"
                    with door_dir in_to;
 ENDIF;
```

2. At the start of your game, define WITHOUT_DIRECTIONS:

```
Constant Story "MYGAME1";
Constant Headline "^My first Inform game.^";
Constant WITHOUT_DIRECTIONS;
```

3. Between the **Includes** of Parser.h and VerbLib.h, place these lines:

```
Include "Parser";

!-------------------------------------------------------------------------!
Class  CompassDirection
  with number 0,
       description [;
           if (location provides compasslook && location.compasslook(self))
               rtrue;
            print "You see nothing special ";
            if (self ~= u_obj or d_obj) print "to the ";
            print_ret (name) self, ".";
            ],
        compasslook 0,  ! use it somewhere just to satisfy the compiler
  has   scenery;

Object  Compass "compass" has concealed;

CompassDirection -> n_obj "north"      with door_dir n_to,
                    name 'n//' 'north';
CompassDirection -> s_obj "south"      with door_dir s_to,
                    name 's//' 'south';
```

```
CompassDirection -> e_obj "east"       with door_dir e_to,
                    name 'e//' 'east';
CompassDirection -> w_obj "west"       with door_dir w_to,
                    name 'w//' 'west';
CompassDirection -> ne_obj "northeast" with door_dir ne_to,
                    name 'ne' 'northeast';
CompassDirection -> nw_obj "northwest" with door_dir nw_to,
                    name 'nw' 'northwest';
CompassDirection -> se_obj "southeast" with door_dir se_to,
                    name 'se' 'southeast';
CompassDirection -> sw_obj "southwest" with door_dir sw_to,
                    name 'sw' 'southwest';
CompassDirection -> u_obj "above"      with door_dir u_to,
                    name 'u//' 'up' 'above' 'ceiling' 'roof' 'sky';
CompassDirection -> d_obj "below"      with door_dir d_to,
                    name 'd//' 'down' 'below' 'floor' 'ground';
CompassDirection -> out_obj "outside"  with door_dir out_to;
CompassDirection -> in_obj "inside"    with door_dir in_to;
!--------------------------------------------------------------------------!

Include "VerbLib";
```

4. After the **Include** of `Grammar.h`, place these lines:

```
Include "Grammar";

Extend 'look'
       * noun=ADirection      -> Examine
       * 'to' noun=ADirection -> Examine;
```

5. In any Room With A View, optionally include a **compasslook** property:

```
Room    study "Your study"
  with  description "There is a doorway to the east of this austere room.",
        compasslook [ obj;
            if (obj == u_obj or d_obj) rfalse;
            if (obj == e_obj) "You see a doorway.";
            "You see the wall.";
            ],
           e_to hallway;
```

How can I embed object details in a room's description?

Normally, Inform prints a room's description, and then lists the objects currently in that room:

```
The farmyard
A nondescript area between the house to the west and a gaping wooden
barn to the east.

You can see a turkey here.
```

It isn't really surprising to find a turkey in a farmyard, so having it listed in a separate paragraph like this has the effect of unnaturally drawing the player's attention to it -- "Aha! If a turkey is called out, it must be important!". Now perhaps the turkey *is* important, but you'd prefer the player to work that out for herself. What you'd like to present is something like this, where the turkey is only mentioned casually as part of the general scene:

```
The farmyard
A nondescript area between the house to the west and a gaping wooden
barn to the east. A turkey scratches in the dust.
```

One way of achieving this is to use the **describe** property. Here's a fragment of the game:

```
Room    farmyard "The farmyard"
  with  description [;
            print "A nondescript area between the house to the west
                   and a gaping wooden barn to the east.";
            if (turkey in self) print " A turkey scratches in the dust.";
            new_line;
            ],
        e_to barn,
        w_to farmers_house;

Room    barn "Wooden barn"
  with  description [;
            print "The farmyard opens to the west of this cluttered old
                   building.";
            if (turkey in self) print " A turkey pecks at scattered straw.";
            new_line;
            ],
        w_to farmyard;

Room    farmers_house "Farmer's house"
  with  description "You're in a house.",
        e_to farmyard;

Object  turkey "turkey" farmyard
  with  name 'turkey',
        description "Nice and plump -- Thanksgiving must be near.",
        describe [;
            if (self in farmyard or barn) rtrue;
            "^A turkey hops around.";
            ];
```

More information in the DM: §14 §26

In rooms where you want the turkey to blend in (such as the Farmyard and Barn), do two things:

1. append an appropriate throwaway turkey reference to the room's **description** *if* the turkey is actually present (don't forget the leading space).
2. include that room in the turkey's **describe** property in the list of locations which simply **rtrue** (which causes the turkey *not* to be listed separately).

In other rooms (such as the Farmer's house) do nothing; if the turkey should stray there, it will be listed separately as usual.

Is it possible to disable TAKE ALL?

Many authors dislike the Library's built-in support for TAKE ALL, for a couple of reasons: it generates foolish messages by attempting to take **scenery** and **static** objects, and it reveals objects whose presence the author may not wish to make readily apparent.

Processing of ALL is handled by the grammar tokens **multi**, **multiheld**, **multiexcept** and **multiinside**. One way of disabling ALL is to re-write the relevant routines in `Parser.h`, but this would be a mammoth task, far far too complex to describe here. Another way is to change the Library's grammar definitions -- in `Grammar.h` or by using **Extend...replace** directives -- to change the **multi** tokens to the simpler **noun** and **held**. However, this is not a good idea, because it would remove also the ability to list nouns explicitly: TAKE BELL, BOOK AND CANDLE would have to become TAKE BELL. TAKE BOOK. TAKE CANDLE.

So here's a third way. The **ChooseObjects()** entry point allows you to influence which objects are included by an ALL. It's easy enough to specify that for the action ##Take (triggered by, for example: GET ALL, PICK UP ALL, REMOVE ALL, TAKE ALL), the ALL simply doesn't match anything, though for other actions it works as before. Here's the code:

More information in the DM: §33

For a more sophisticated solution, look at takeall.inf from Kory Heath and Lucian Smith

```
[ ChooseObjects obj code
        retval;
        obj = obj;              ! Avoid a compiler warning
        switch (code) {
            0:                  ! Parser is excluding obj from ALL
                ;               ! ... accept parser's decision
            1:                  ! Parser is including obj in ALL
                if (action_to_be == ##Take) retval = 2;
                                ! ... force exclusion if TAKE; accept otherwise
            2:                  ! Parser is asking for 'appropriateness' hint
                ;               ! ... but we don't provide one
            }
        return retval;
        ];
```

This works fine, though the fact that TAKE ALL now generates the standard message "There are none at all available!" irrespective of how many objects are in scope is awkward and misleading. So let's fix that, by defining a **LibraryMessages** object:

More information in the DM: §25

```
Include "Parser";

Object  LibraryMessages
  with  before [;
            Miscellany:
                if (lm_n == 44 && action_to_be == ##Take)
                    "You can't use ALL in this context.";
            ];

Include "VerbLib";
```

where we intercept Miscellany message 44 -- "There are none at all available!" -- and, for a ##Take action, substitute a more appropriate replacement.

Can I avoid printing "(which is empty)" after a container?

This topic has been superseded by libray 6/11

When Inform is listing the objects in a room -- "You can see a rusty axe and some blood here." -- it provides additional information about containers and supporters. For example, you might encounter "You can see a fish tank (in which is a brick (on which is a bottle (which is empty))) here.", not perhaps the ideal form of presentation.

It turns out that none of the standard object properties will change those parenthetical clauses. **invent** *ought to* -- "This routine is for changing an object's inventory listing" -- and it does indeed provide exactly the desired control when displaying the player's possessions, but it has no effect when listing the objects in a room. While this is not necessarily a bug, you may well wish to alter the default behaviour; here's one way.

Unfortunately, the fix requires a change to the standard list-maker **WriteListFrom()**; this routine creates all object lists printed by the library, and its behaviour is controlled by a series of 'style' bits. During an player inventory, the library calls **WriteListFrom()** with the FULLINV_BIT style setting, while the call to list a room's contents includes the PARTINV_BIT style setting. The problem is that the code which deals with FULLINV_BIT takes note of an object's **invent** property (if it has one), while the code for PARTINV_BIT does not.

To make the PARTINV_BIT code recognise an optional **invent** property, small modifications are needed to two library routines: **WriteBeforeEntry()** and **WriteAfterEntry()**. Rather than edit `verblibm.h` directly -- messing with the library is always a somewhat traumatic activity -- you can use my complete ready-to-run replacements for the two routines. Simply download Roger Firth's `WriteList.h` from the Archive, and **#Include** it in your game *between* `Parser.h` and `VerbLib.h`. That's all.

If an object has an **invent** property, it will be invoked in the usual way (with inventory_stage first set to 1, and then set to 2) both when mentioning that object in a room description, and when listing it in the player's inventory. By default you'll get the same output each time. If you need to distinguish between the two occasions, you can test (c_style&PARTINV_BIT) -- **true** during a room description -- or (c_style&FULLINV_BIT) -- **true** during an inventory. Here's an example:

More information in the DM: §26 §27

```
Include "Parser";
Include "WriteList";
Include "VerbLib";

Object  -> "sack"
  with  name 'sack',
        invent [;
            ! When listing objects in the player's inventory
            if (c_style&FULLINV_BIT) rfalse;

            ! When listing objects at the end of a room description
            if (inventory_stage == 1) switch (children(self)) {
                0: print "an empty sack";
                1: print "a sack containing ", (a) child(self);
                default: print "an assortment of objects in a sack";
                }
            rtrue;
            ],
  has   container open;
```

Can I avoid printing "(the *objectname*)" after certain commands?

One feature of the Inform parser is its ability -- in the right circumstances -- to infer which object the player intended. For example:

```
>INV
You are carrying:
  a red ball
  a blue ball

>DROP
What do you want to drop?

>BALL
Which do you mean, the red ball or the blue ball?

>RED
Dropped.

>DROP
(the blue ball)
Dropped.
```

In first part of this example, the parser handles the ambiguity of which ball to drop. In the second part, when there is no longer any ambiguity, the parser makes an inference -- that the blue ball must be the intended object -- and drops it without further prompting. But not without telling you what it's doing; hence you see "(the blue ball)" displayed.

Mostly, this is helpful, or at worst harmless. Just occasionally, it seems unnecessary and intrusive (for instance, in the discussion on water below -- on such occasions, you might well wish to turn the feature off):

```
>POUR WATER INTO BARREL
(the water into the barrel)
You pour the water into the barrel.
```

Sadly, there's no easy way. The heart of the parser, defined in parserm.h, is an 800-line routine called **Parser__parse()**. Three-quarters of the way through, this is the code responsible for the inference message:

```
if (inferfrom ~= 0) {
    print "("; PrintCommand(inferfrom); print ")^";
}
```

where **inferfrom** is a global variable set (mainly) when another library routine -- **NounDomain()** -- has determined (a) that the player hasn't specified an object unambiguously, and (b) that this doesn't matter, because it can infer what must have been intended.

So, your options are to edit parserm.h and comment-out those three lines, to Replace **Parser__parse()** with your own variant which omits the lines... or to live with the situation. I recommend this last course of action.

How does the list-maker work?

The Library includes a routine for displaying lists of objects, used primarily by LOOK (to present the visible objects in the current room) and by INVENTORY (to present the player's possessions). The routine is described in the DM4, but the information on how best to employ it is frustratingly brief: "the best way is to experiment" is Graham's none-too-helpful advice. The routine is this:

More information in the DM:§27

```
WriteListFrom(object, style, depth);
```

where *object* is the first object to be listed, usually **child(location)** or **child(player)**, *style* is a magic number defining the form of the required listing, and *depth* is an undocumented value, usually 0 but sometimes 1 (or higher?). It's the business of getting the *style* number right -- by combining control bits -- which is the tricky part. There are thirteen defined bits, capable of being combined in over 8000 ways; fortunately, the vast majority of the combinations don't do anything distinctive, so that we can reduce the problem to something manageable. I can identify only eleven basic styles, arbitrarily labelled A-K, which we'll explain by example. Here's an 'interesting' collection of objects:

```
Object  "tray" selfobj          has supporter;
  Object  -> -> "bottle"        has container;
    Object  -> -> -> "wine"     with article "some";
  Object  -> -> "glass"         has container open;
    Object  -> -> -> "drop of water";
  Object  -> -> "dish"          has supporter;
    Object  -> -> -> "tablet";

Object  "box" selfobj           has container openable open;
  Object  -> -> "bell";
  Object  -> -> "book"          has container openable lockable locked;
  Object  -> -> "candle"        has light;
```

and here's how **WriteListFrom()** describes them:

Style	*style* bits	Result
A	ENGLISH_BIT	a tray and a box
B	ENGLISH_BIT+RECURSE_BIT	a tray, on top of which are a bottle, a glass, inside which is a drop of water and a dish, on top of which is a tablet and a box, inside which are a bell, a book and a candle

C	ENGLISH_BIT+RECURSE_BIT+PARTINV_BIT	a tray, on top of which are a bottle (which is closed), a glass, inside which is a drop of water and a dish, on top of which is a tablet and a box, inside which are a bell, a book (which is closed) and a candle
D	ENGLISH_BIT+RECURSE_BIT+PARTINV_BIT+TERSE_BIT	a tray (on which are a bottle (which is closed), a glass (in which is a drop of water) and a dish (on which is a tablet)) and a box (in which are a bell, a book (which is closed) and a candle)
E	ENGLISH_BIT+RECURSE_BIT+FULLINV_BIT	a tray, on top of which are a bottle, a glass, inside which is a drop of water and a dish, on top of which is a tablet and a box (which is open), inside which are a bell, a book (which is closed and locked) and a candle (providing light)
F	ENGLISH_BIT+RECURSE_BIT+FULLINV_BIT+TERSE_BIT	a tray (on which are a bottle, a glass (in which is a drop of water) and a dish (on which is a tablet)) and a box (which is open) (in which are a bell, a book (which is closed and locked) and a candle (providing light))
G	NEWLINE_BIT	a tray a box
H	NEWLINE_BIT+RECURSE_BIT	a tray a bottle a glass a drop of water a dish a tablet a box a bell a book a candle

I	NEWLINE_BIT+RECURSE_BIT+INDENT_BIT	a tray
		    a bottle
		    a glass
		      a drop of water
		    a dish
		      a tablet
		  a box
		    a bell
		    a book
		    a candle

J	NEWLINE_BIT+RECURSE_BIT+INDENT_BIT+PARTINV_BIT	a tray
		    a bottle (which is closed)
		    a glass
		      a drop of water
		    a dish
		      a tablet
		  a box
		    a bell
		    a book (which is closed)
		    a candle

K	NEWLINE_BIT+RECURSE_BIT+INDENT_BIT+FULLINV_BIT	a tray
		    a bottle
		    a glass
		      a drop of water
		    a dish
		      a tablet
		  a box (which is open)
		    a bell
		    a book (which is closed and locked)
		    a candle (providing light)

Notes:

- All styles also accept **+ISARE_BIT** (which displays "is" or "are" at the front of the list).
- All styles also accept either **+DEFART_BIT** (which displays "the" instead of "a" before each object), or **+NOARTICLE_BIT** (which omits "a" altogether).
- All styles also accept **+CONCEAL_BIT** (which ignores an object with a **concealed** or **scenery** attribute).
- All styles also accept **+WORKFLAG_BIT**. When *depth*=0, this ignores an object unless it (or one of its parents) has a **workflag** attribute. When *depth*>0, this has no effect. In theory you can apply this attribute yourself to various objects in order to create a customised list; in practice the **workflag** attribute is primarily for use by the Library and thus prone to being set and cleared; you'll need to establish your settings immediately prior to calling **WriteListFrom()**.
- In styles H, I, J and K, you can replace **+RECURSE_BIT** (which ignores a child object unless its parent has a **supporter**, or **transparent**, or **container** plus **open** attribute) by **+ALWAYS_BIT** (which always includes it).
- In styles G, H, I, J and K, the display ends with a newline.
- Styles B, C and E misrepresent the inventory, by implying that objects are erroneously in or on previously-listed objects; see the Inform Patch List. The use of **+TERSE_BIT** removes the ambiguity.
- A normal INVENTORY [TALL] command uses style K; an INVENTORY WIDE uses style E.
- LOOK uses style D (as part of messages 4, 5 and 6), as does the **Locale()** Library routine.
- OPEN uses style A (as part of message 4), and SEARCH uses style A (as part of messages 3 and 7).

Occasionally you may wish to override the prevailing style of listing, for example by having a 'tall' inventory display the contents of one item on a single line. You can create your own list-writer to do this, by using an object's **invent** property. However, you can only reliably use **WriteListFrom()** *within an **invent** routine* if you obtain the specially-modified version of **WriteListFrom()** which is included in `WriteList.h` (described in the previous topic).

For example, to simplify the presentation of the box -- by mentioning only the immediate child objects inside it while ignoring any lower-level children, and also by omitting descriptive comments like "(providing light)" -- you could try this:

```
Object  "box" selfobj
  with  invent [;
            if (inventory_stage == 2 && child(self)) {
                print " (in which";
                WriteListFrom(child(self), ENGLISH_BIT+ISARE_BIT);
                print ")";
                rtrue;
            }
        ];
  has   container openable open;
```

This enhancement takes effect in any style which uses **PARTINV_BIT** or **FULLINV_BIT** (and so affects room descriptions as well as inventories); for example style C now produces "a tray, on top of which are a bottle (which is closed), a glass, inside which is a drop of water and a dish, on top of which is a tablet and a box (in which are a bell, a book and a candle)" and style K gives:

```
a tray
  a bottle
  a glass
    a drop of water
  a dish
    a tablet
a box (in which are a bell, a book and a candle)
```

Incidentally, here's a tiny routine which walks the object tree, starting from a given object, in exactly the same way as **WriteListFrom()**. For each object *obj* that **RunRoutineFrom()** finds, it calls *R*(*obj*):

```
[ RunRoutineFrom o R
    obj;
    objectloop (obj from o) {
        R(obj);
        if (child(obj)) RunRoutineFrom(child(obj), R);
    }
];
```

These two code fragments produce the same output:

```
[ PrintIt obj; print_ret (a) obj; ];
RunRoutineFrom(myObj, PrintIt);

WriteListFrom(myObj, NEWLINE_BIT+ALWAYS_BIT);
```

Could you explain what "in scope" means?

Scope is one of those topics which can be as easy or as complex as you care to make it. Let's try to make it easy.

More information in the DM: §32

Put simply, what's "in scope" -- roughly, the things you can EXAMINE -- is represented by the contents of the room you're currently in: yourself, your possessions, and any other objects in that particular room. By convention, even in an external location the surrounging walls are opaque, preventing you from seeing objects in the adjacent rooms.

Almost all of the time, Inform takes care of scope by itself. Providing that you've set your object attributes correctly, you'll see fixed objects and portable objects, supporters and any objects on them, containers and any objects in them (but only if the container is open), cats and dogs and NPCs. Further, you can usually TOUCH, TASTE, TAKE any object (unless it's locked inside a glass case), and you need only some very simple **before** properties to deal with the obvious exceptions: objects which are in the distance, repulsive, ludicrously heavy, and so on. The point is: the library does an excellent job of managing scope, so that you usually get the desired effect with very little effort.

Just occasionally, though, you need to extend the standard scope, by enabling the player to mention an object which *isn't* clearly lying around in the room. There are two main techniques available: you can hide the requisite object inside the room by some means, or you can tell the interpreter to look for it somewhere else entirely.

Smuggling extra objects into the room. We're now assuming that you want the player to be able to refer to objects even though they're not listed as part of the room's contents. Your options include:

- The **concealed** attribute. Such objects don't get listed but can be TAKEn if the player knows they're there. Also...
- The **scenery** attribute. Such objects don't get listed, and can't be TAKEn. Actually, these differences are largely academic, because you'd be unlikely to permit TAKE on objects of this type. Here's an example, which might be polluting a single room or, by means of a **found_in** property, be detectable in several locations:

```
Object  "awful smell"
  with  name 'awful' 'smell' 'stench' 'stink' 'odour',
        react_before [;
            Smell: if (noun == nothing) <<Smell self>>;
        ],
        before [;
            Smell,Taste: "Ugh!";
            default: "It's hard to do that to a smell.";
        ],
  has   concealed;
```

- Containment within a **transparent** object. Such objects don't get listed and can't be TAKEn, but do move around with their parent if that's portable. For example:

```
Object  "remote control"
  with  name 'remote' 'control',
        description "It has only one button.",
  has   transparent;

Object  -> "button"
  with  name 'button',
        before [;
            Push: if (self has on) {
                give self ~on; "The distant hum stops.";
            }
            else {
                give self on;  "You hear a distant humming sound.";
            }
        ],
  has   ~on;
```

- The **add_to_scope** property. This property comprises either a list of one or more additional objects, or a routine which makes one or more calls to **AddToScope(***object***)**. For example:

```
Object  newSelfobj "yourself"
  with  description "You don't much care for what you see.",
        number 0,
        add_to_scope hair,
  has   animate concealed proper transparent;

Object  hair "your hair"
  with  name 'hair' 'head',
        before [;
            Examine,Pull,Rub,Search,Touch:
                "You scratch your head; dandruff showers.";
            default: "Huh?";
        ],
  has   proper;
```

More information in the FAQ topic: How can I reconfigure the Player Character (PC)?

Note: the DM4 is confused on whether an **add_to_scope** routine should call **AddToScope()** or **PlaceInScope()**; the consensus is to use **AddToScope()**.

Looking for objects elsewhere. The second possibility is rather less common than the first, but it's the best approach to one specific problem. The grammars for most verbs which handle objects are similar to this example:

```
Verb 'open' 'unwrap' 'uncover' 'undo'
    * noun              -> Open
    * noun 'with' held  -> Unlock;
```

More information in the DM: §31 §32

The **noun** token matches any object in scope (usually, in the room); the **held** token matches any object carried by the player. That makes sense: you can't expect to OPEN THE COFFIN unless it's there in front of you. However, what if you want to ASK UNDERTAKER ABOUT COFFIN? It seems a reasonable question, and one that the undertaker should be able to answer, even though the object itself isn't currently visible. The standard grammar here is:

```
Verb 'ask'
  * creature 'about' topic    -> Ask;
```

but the **topic** token is matched by any text at all, leaving you with the messy task of parsing it for meaningful content. An better approach is to let the Inform parser do the hard work for you. You need four items:

1. A set of 'objects', not physically present in the game, whose only role is to define the words you can ask about in their **name** properties:

```
Object  AskTopics "topics for ASK";
Object  -> t_coffin "coffin"
     with name 'coffin' 'box' 'casket';

Object  -> t_grave  "grave"
  with name 'grave' 'hole';

Object  -> t_stone  "gravestone"
  with name 'gravestone' 'stone' 'monument';
etc...
```

2. A routine to look for those words:

```
  [ AskTopicsScope;
      switch (scope_stage) {
       1: rfalse;
       2: ScopeWithin(AskTopics); rtrue;
       3: "At the moment, even the simplest questions are confusing.";
      }
];
```

3. A new grammar which calls that routine:

```
Extend 'ask' replace
    * creature 'about' scope=AskTopicsScope   -> Ask;
```

4. And lastly, an NPC **life** property to supply the answer which matches the 'object' parsed into the **second** variable:

```
Object  undertaker "undertaker"
  with  ...
        life [;
          Ask: switch (second) {
            t_coffin: "~Best Brazilian mahogany -- it'll last a
                       lifetime.~";
            t_grave:  "~Six feet deep, and not an inch less.~";
            t_stone:  "~We can offer you granite, slate or marble.~";
            ...
            default:  "~I'm so sorry; I can't help you with that.~";
          }
        ],
  has   animate;
```

For more information on conversation with NPCs, see Roger Firth's InFact pages

The reason we're presenting this here is because of that **scope=** token, which uses a routine -- written by you, but almost always of the form shown -- to temporarily redefine the current scope. **ScopeWithin()** is a library routine which places the contents of the specified object in scope, and here it's used to bring the conversational topic objects into scope, and so enable the conversation to refer to as many subjects as you care to define.

Checking on scope. An object sometimes needs to check for itself whether it's in scope. Typically this is within a daemon or timer; for example, if the player removes a supporting column, he may have two or three turns to leave the room before the ceiling falls in and -- if he's still there -- kills him. Here are three simple tests:

- **if (self in location) ...** verifies that the object is in the player's current room;
- **if (IndirectlyContains(real_location,self)) ...** does the same thing, but also allows for the object being inside a container, and for the room being dark;
- **if (TestScope(self)) ...** is usually the best approach, since it takes care of the full range of scope testing (but be careful: an exploding bomb should still kill the player even if it's out-of-scope within a closed paper bag).

Remembering a lost object. Finally, an exploration of techniques for dealing with an out-of-scope object whose previous presence the player might well recollect. Consider this example:

```
> DRINK WATER
You drink the water, and it's all gone.

> DRINK WATER
You can't see any such thing.
```

True enough, and easy to program. But what if you're after this effect?

```
> DRINK WATER
You drink the water, and it's all gone.

> DRINK WATER
What water?
```

Here are some different ways of achieving this.

```
Object  -> water "water"
  with  name 'water' 'liquid',
        article "some",
        description "Very cold and very wet ",
        before [;
          Drink:
            move fake_water to parent(self); remove self;
            "You drink the water, and it's all gone.";
        ];

Object  fake_water "water"
  with  name 'water' 'liquid',
        article "some",
        react_before [;
            if (self == noun or second) "What water?";
        ],
  has   scenery;
```

These first four solutions all use some slight trickery in order to keep a fake object with the name 'water' in scope, even after the real water has been consumed. That way, the command DRINK WATER will continue to be accepted by the parser; it's the fake water's job to react to all commands with the "What water?" message.

In this example, the first DRINK WATER command removes the real water object and replaces it with a fake equivalent. The fake then rejects all subsequent commands to DRINK, EXAMINE, or do anything at all with the missing water.

```
Object  -> water "water"
  with  name 'water' 'liquid',
        article "some",
        description "Very cold and very wet ",
        before [;
          Drink:
            give self scenery;
            "You drink the water, and it's all gone.";
        ],
        react_before [;
            if (self has scenery && self == noun or second)
                "What water?";
        ],
  has   ~scenery;
```

This example is very similar, but rather than replacing one object by another, it uses the **scenery** attribute to transform the real water into fake.

One advantage of this approach is that pronouns still work properly; for example, you can use these three commands successfully, because IT still refers to the water after it's been consumed:

EXAMINE WATER
DRINK IT
DRINK IT

```
Object  -> water "water"
   with  name 'water' 'liquid',
         article "some",
         description "Very cold and very wet ",
         before [;
           Drink:
             remove self; MoveFloatingObjects();
             "You drink the water, and it's all gone.";
         ];

 Object  fake_water "water"
   with  name 'water' 'liquid',
         article "some",
         react_before [;
             if (self == noun or second) "What water?";
         ],
         found_in [;
             if (TestScope(water) == false) rtrue;
         ],
   has   scenery;
```

The previous two solutions work only in a single location; move to another room, and your quest for water reverts to the standard "You can't see any such thing.". Here's a way of using the **found_in** property so that you'll obtain the specialized "What water?" response anywhere.

```
 Object  -> water "water"
   with  name 'water' 'liquid',
         article "some",
         description "Very cold and very wet ",
         before [;
           Drink:
             remove self;
             "You drink the water, and it's all gone.";
         ];

 Object  fake_water "water"
   with  name 'water' 'liquid',
         article "some",
         react_before [;
             if (self == noun or second) "What water?";
         ],
   has   scenery;

 Object  mySelfobj "(self object)"
   with  short_name [; return L__M(##Miscellany, 18); ],
         description [; return L__M(##Miscellany, 19); ],
         before NULL, after NULL, life NULL, each_turn NULL,
         time_out NULL, describe NULL,
         capacity 100, parse_name 0,
         orders 0, number 0,
         add_to_scope [;
             if (parent(water) == nothing)
                 AddToScope(fake_water);
         ],
   has   concealed animate proper transparent;
```

This example achieves the same thing by a different technique; it uses an **add_to_scope** property on the player object, so that the fake water is brought into scope as the player moves around.

Unfortunately, you can't just assign a new **add_to_scope** property value to **selfobj**, because the standard Library object doesn't provide an initial value for you to overwrite. Instead, you must define your own version of **selfobj** -- based on the Library object -- with the addition of an appropriate **add_to_scope** property. Also, you must add this line to your **Initialise()** routine:

```
  player = mySelfobj;
```

```
  Object  -> water "water"
    with  name 'water' 'liquid',
          article "some",
          description "Very cold and very wet ",
          before [;
            Drink: remove self;
                   "You drink the water, and it's all gone.";
          ];

  [ ParseWater n;
      while (NextWord() == 'water' or 'liquid') n++;
      if (n == 0) return GPR_FAIL;
      wn--;
      return GPR_PREPOSITION;
  ];

  [ WhatWaterSub; "What water?"; ];

  Extend 'drink'
      * ParseWater -> WhatWater;
```

To finish with, here are two quite different lines of attack. In this example we define a 'general parsing routine' (DM4 §31) specifically to look for 'water' and 'liquid', and we also append a line to the DRINK verb's grammar. The effect is that, if DRINK WATER isn't matched by a real water object, it'll still match this new grammar line, and thus trigger the WhatWater action to produce the required message.

```
  Object  -> water "water"
    with  name 'water' 'liquid',
          article "some",
          description "Very cold and very wet ",
          before [;
            Drink:
              remove self;
              "You drink the water, and it's all gone.";
          ];

[ ParserError error_type
      x;
      for (wn=1,x=0 : wn<=parse->1 : wn++)
          if (WordInProperty(parse-->(2*wn-1), water, name)) x++;
      if ((error_type == CANTSEE_PE && x) ||
          (error_type == ITGONE_PE  && PronounValue('it') == water))
          "What water?";
      rfalse;
  ];
```

This example steps in at the last possible moment, just as the parser is about to report "You can't see any such thing." and, if the parse error relates to (what is probably) the missing water, displays our customized response instead.

What's the easiest way to shine light everywhere?

More information in the DM: §19

One of the first things you learn, when starting out to design an Inform game, is that your rooms will be pitch dark unless a source of light is available. That's what the **light** attribute does for you, and it's most commonly applied to each room:

```
Object  hallway "Dingy hall"
  with  description "Steps lead down into darkness.",
        e_to front_door,
        w_to kitchen,
        d_to cellar,
  has   light;

Object  kitchen "Small kitchen"
        ...
  has   light;
```

However, (with luck) you'll quickly realize that a Room class simplifies things a little:

```
Class    Room
  with  description "A bare and featureless room.",
  has   light;

Room    hallway "Dingy hall"
  with  description "Steps lead down into darkness.",
        e_to front_door,
        w_to kitchen,
        d_to cellar;

Room    kitchen "Small kitchen"
        ...;
```

More information in the FAQ topic: What does class inheritance do for me?

That approach is both easy and flexible: your rooms will now be illuminated unless you say otherwise. If you don't need that flexibility -- if darkness isn't a factor anywhere in your game -- then you can make things even simpler. Here are three ways:

Providing the sun. Use an object like this:

```
Object with found_in [; rtrue; ], has light scenery;
```

Lighting up the player. Add this line to Initialise():

```
give player light;
```

Bending the rules. Change the library routine **OffersLight()**:

```
Replace OffersLight;

[ OffersLight obj; if (obj) rtrue; else rfalse; ];
```

Why is water so difficult to model?

More information in the DM: §50

Given how commonly they occur in real life, the implementation in a game of water-like liquids is a remarkable challenge. TAKE is difficult (you need a suitable container, and what happens to any existing contents?), as is DROP (what happens to the spillage?). Worse, a liquid can be divided and combined indefinitely, and mixed with other liquids. Or solids. If there's a bit of it, you could DRINK it; if there's a lot of it, SWIM becomes a possibility; and so on.

See Jim Fisher's **ORliquid.h** and Emily Short's amazingly comprehensive (and very large) **WaterElement.h** in the Archive

There are a couple of excellent library extensions in the Archive to deal with liquids, but for a game where a bucket of water has only an insignificant walk-on role, you might fancy something rather simpler. So here are a couple of object classes to act as a starting point. You get support for FILL and EMPTY, and for pouring the contents of one container into another. You *don't* get support for specific quantities of liquid -- a container is either empty or full -- nor for mixing liquids, nor for anything else beyond the basics. And even so, there's quite a lot of code.

More information in the DM: §3.11

First, the Water class can be used for the liquid object itself. It enables you to create and delete instances of the class at run-time, redirects several possible actions to the liquid's container, and includes an **infinite** property to distinguish between finite (the contents of a bottle or glass) and infinite (a lake, a fountain) amounts:

```
Class   Water(2)
  with  name 'water',
        short_name "water",
        article "some",
        infinite false,     ! is this an inexhaustible source of liquid?
        before [;
          Drink:
            "A small sip refreshes you.";
          Take:
            "[Use FILL and the name of a suitable container.]";
          Insert,Transfer:
            if (parent(self) ofclass Bucket)
              <<EmptyT (parent(self)) second>>;
            if (second ofclass Bucket)
              <<Fill second>>;
            "I'm not sure that's a suitable container.";
          Drop,Empty,EmptyT:
            if (parent(self) ofclass Bucket)
              <<(action) (parent(self)) second>>;
        ],
        react_before [;
          Fill:
            if (noun ofclass Bucket) rfalse;
            "I'm not sure that's a suitable container.";
        ];
```

The Water class is heavily inter-dependent on the Bucket class, which defines an object capable of containing the liquid; it doesn't have to be a bucket -- a glass, jar, bottle or barrel would all be appropriate. Most of the work is handling the actions of Fill, Empty and EmptyT (EMPTY x INTO y) for Water objects, ensuring that Water isn't mixed with other objects:

```
 Class   Bucket
   with  before [ x y; x = child(self);
           Empty:
             if (~~x ofclass Water) rfalse;     ! follow standard action
             Water.destroy(x);
             print_ret (The) x, " pours onto the ground and disappears.";
           EmptyT:
             if (~~x ofclass Water) rfalse;     ! follow standard action
             if (second ofclass Bucket) {
                 y = child(second);
                 if (~~y) {                     ! second is empty
                     move x to second;
                     "You pour ", (the) x, " into ", (the) second, ".";
                 }
                 if (y ofclass Water)
                     print_ret (The) second, " is already full of ",
                               (name) y, ".";
                  else
                     print_ret (The) second, " has other stuff in it.";
             }
             else {
                 if (second == d_obj) <<Empty self>>;
                 "I'm not sure that's a suitable container.";
             }
          Fill:
              if (~~x) {                    ! is there a water source in scope?
                  objectloop (x ofclass Water && x.infinite && TestScope(x)) {
                      x = Water.create(); move x to self;
                      "You fill ", (the) self, " with ", (name) x, ".";
                  }
                  rfalse;                   ! no water available
              }
              if (x ofclass Water)
                  print_ret (The) self, " is already full of ", (name) x, ".";
              else
                  print_ret (The) self, " has other stuff in it.";
            Receive:
              if (x ofclass Water)
                  print_ret (The) self, " is already full of ", (name) x, ".";
          ],
    has   container open;
```

If you FILL a Bucket from a source of Water, there are two Water objects in the room. A reference to 'WATER' seems to favour the infinite source rather than the finite amount in the Bucket, which feels the wrong way round. So we can define a **ChooseObjects()** routine to rectify this:

More information in the DM: §33

```
  [ ChooseObjects obj code;
      if (code == 2 && obj ofclass Water)
          if (obj.infinite) return 1; else return 2;
  ];
```

Finally, here are a couple of useful extensions to the standard grammar:

```
  Verb 'pour' = 'empty';

  [ Wet; if (noun ofclass Water) rtrue; rfalse; ];

  Extend 'fill'
      * noun 'with' noun=Wet      -> Fill
      * noun 'at'/'from' noun     -> EmptyT reverse;
```

The nice thing about defining object classes, however complex, is that the actual objects themselves are pretty simple. Here's a bubbling fountain, from which you can fill a jar with water:

```
Object  -> fountain "fountain"
  with  name 'little' 'stone' 'fountain' 'pool',
        description
            "Water gushes from the little stone fountain
             and collects in the pool beneath.",
        before [;
          EmptyT:
            if (second ofclass Bucket) <<Fill second>>;
            "I'm not sure that's a suitable container.";
        ],
  has   static transparent;

Water   -> -> "water"
  with  infinite true;

Bucket  -> "jar"
  with  name 'jar' 'pot';

Bucket  -> "barrel"
  with  name 'barrel' 'cask';
```

And this is what you get for your money:

```
>LOOK

A valley in the mountains
The Alpine vista is glorious.

You can see a fountain and a barrel (which is empty) here.

>EXAMINE FOUNTAIN
Water gushes from the little stone fountain and collects in the pool
beneath.

>FILL JAR
You fill the jar with water.

>INV
You are carrying:
  a jar
    some water

>POUR WATER INTO BARREL
(the water into the barrel)
You pour the water into the barrel.

>LOOK

A valley in the mountains
The Alpine vista is glorious.

You can see a fountain and a barrel (in which is some water) here.

>EMPTY BARREL
The water pours onto the ground and disappears.
```

How does everybody know where the north is?

More information in the DM: §9

Once in a while there comes a player or designer complaining that the use of cardinal directions in a game seems awkward and absurd. Awkward, because it doesn't read very elegantly in room descriptions to have a list of available exits which mention "north, south and southwest" -- it may break the mood of the narration, especially if you are indoors. Absurd, because games assume that players have an innate and accurate sense of direction, a compass inside their brains, which lets them assess that, for example, the banana lies to the northeast, even when they are prisoners in an underground cave. Wouldn't it be more logical to use relative directions -- "to your left", "ahead of you", and so on -- and be done with?

Many discussions arise from the "Compass versus Relative directions" debate. Not so long ago, Mike Roberts, designer of the excellent Text Adventure Development System, decided to make a TADS test game, Rat In Control, which allowed players to switch between both systems; the idea was to see if the conventional Compass Directions system was in fact clearer and easier to use, and if players would form confused perceptions of the map when using relative directions. You may check the announcement and further discussion of the experiment here.

Disregarding the question of use from a player's perspective, there are some design issues which makes the implementation of relative directions a difficult task. Let's try to define the problem and its ramifications.

Imagine our player standing in a square room, looking to the north. We need to describe the new directions, so a start would be "ahead of you", "to your right", "to your left" and "behind you". However, things begin to sound a bit more clumsy with diagonal directions: "diagonally ahead of you and to your right", or "slightly to your left", or "over your right shoulder" or (heaven forbid) "round to your left and a bit more". You can even complicate things further by adding the third dimension: "above, diagonally ahead of you and to your right" (or perhaps just "two o'clock high"), but let's presume for the sake of simplicity that your game doesn't need those.

So our player is in a square room facing north, which, for the time being, has become "ahead of you". When we use compass directions, our movement verbs echo the direction we want to go: if we type EAST, the player moves in that direction and arrives at a new location; end of the problem. With relative directions, however, you have to take in consideration which way the player is facing, because if you type AHEAD or some such, it will get you to a different place if you're facing to the north or if you're facing to the south. Two problems arise from this:

- the player must be able to rotate, to turn around; we must implement new "turn to the right" type actions, and we must make a decision: how many degrees will the player turn for each action? With eight horizontal directions, 45 degrees might be logical in order to allow the player to face whichever possible direction there is. However, perhaps this is too complicated, and simple quarter turns would be enough.

- If you want the PC to go LEFT, does he scuttle sideways like a crab or, more logically, does he turn to face the new direction and then walk forward? The immediate consequence would be that the player is now facing to a different direction when he arrives at the new location, and what was "left" has become "ahead". If you don't want to allow for this complication, you could perhaps suppose that the PC travels to his new destination and then turns again to face to the north -- meaning that he *does* have a compass in his head after all -- but then you would have to explain this behaviour in some way, or players might not understand what's going on.

And then, of course, you still have the little problem of room descriptions. Consider the following paragraph:

```
The Library
Bookshelves cover all the walls in this magnificent room. There's
an arch to your left leading into the living room, and double oak
doors ahead of you connect with your private office.
```

So far, so good. But now, suppose that we enter our private office, which proves to be a dead end, and then return to the library. Unless we always force the player to face to the north (which is really a bit silly), we now find that the arch is "to the right" and the double oak doors "behind you". So your descriptions need to implement some way of catering for these variations.

The following code is a (simplified and crude) example which implements just four relative directions (ahead, right, behind and left), without removing the existing compass directions. You could easily extend it to cover diagonal directions; the system wouldn't change, but you'd be acquiring the malady known as "combinatorial explosion".

```
! Four common properties to add to the new CompassDirection objects:

Property ahead_to;
Property right_to;
Property back_to;
Property left_to;

Global position; ! in which direction is the PC looking?
                 ! 0 = north; 1 = east; 2 = south; 3 = west;
                 ! There are no diagonal directions in this example.

Global op;       ! Absolute position of a room or a fixed object
                 ! in relation to the (adjacent) location where the PC is.
                 ! It doesn't matter how the PC moves, the "office" (e.g.)
                 ! is always to the north of the "library".

! We now need some new Compass Direction objects:

CompassDirection ahead_obj "ahead" Compass
     with door_dir ahead_to, name 'straight' 'ahead' 'forward' 'f//' 'a//';

CompassDirection right_obj "right" Compass
     with door_dir right_to, name 'right' 'r//';

CompassDirection back_obj "behind" Compass
     with door_dir back_to, name 'back' 'backwards' 'behind' 'b//';

CompassDirection left_obj "left" Compass
     with door_dir left_to, name 'left' 'le' 'lf' 'lft';

! The Room class implements the relative directions system.
! Caveat: you cannot use the s_to "can't go" syntax.
! You must use a 'before' property and trap Go actions.

Array cdir --> n_obj e_obj s_obj w_obj;

Class   Room
  with  description "Under Construction.",
        before [;
          Go:
            if (self provides n_to && noun == n_obj) position = 0;
            if (self provides e_to && noun == e_obj) position = 1;
            if (self provides s_to && noun == s_obj) position = 2;
            if (self provides w_to && noun == w_obj) position = 3;
        ],
        ahead_to [; <<Go (cdir-->(position))>>; ],
        right_to [; <<Go (cdir-->((position+1)%4))>>; ],
        back_to  [; <<Go (cdir-->((position+2)%4))>>; ],
        left_to  [; <<Go (cdir-->((position+3)%4))>>; ],
  has   light;
```

```
! Define a printing rule for our changing descriptions:

Array rdir --> "ahead of you" "to your right" "behind you" "to your left";

[ dir obj;
    if (location provides n_to && obj == location.n_to) op = 0;
    if (location provides e_to && obj == location.e_to) op = 1;
    if (location provides s_to && obj == location.s_to) op = 2;
    if (location provides w_to && obj == location.w_to) op = 3;

    if (position == 0 or 2) print (string) rdir-->((position+op)%4);
    if (position == 1 or 3) print (string) rdir-->((position+op+2)%4);
];

! In Initialise, we must define a starting value for the position variable:

[ Initialise;
    location = entrance;
    position = 0;
    ...
];

! And now some grammar:

Array cdir --> "north" "east" "south" "west";

[ TurnRightSub;
    print "You make a quarter turn right. You are now facing to the ";
    position = (position + 1)%4;
    print_ret (string) cdir-->position, ".";
];

[ TurnAroundSub;
    print "You turn around. You are now facing to the ";
    position = (position + 2)%4;
    print_ret (string) cdir-->position, ".";
];

[ TurnLeftSub;
    print "You make a quarter turn left. You are now facing to the ";
    position = (position + 3)%4;
    print_ret (string) cdir-->position, ".";
];

Extend 'go'
    * 'to' noun=ADirection  -> Go;

Extend 'turn' first
    * 'right'               -> TurnRight
    * 'around'              -> TurnAround
    * 'left'                -> TurnLeft;
```

Now, every time you write a room description, all you need to do is use the new printing rule when you describe the positions of adjacent rooms. Supposing there are two rooms named living_room and office respectively, you could write:

```
Room    library "The Library"
  with  description
          "Bookshelves cover all the walls in this magnificent room.
           There's an arch ", (dir) living_room, " leading into the
           living room, and double oak doors ", (dir) office, " connect
           with your private office.",
        ...
```

As you see, the implementation of relative directions is no trivial task, and it's probable that players will find it a bit confusing at run-time. So our advice is that you stick with the conventional Compass directions, which are clean and straightforward, even if they seem a bit awkward and absurd at times.

If you don't want to use the words "north", "south", etc, in your room descriptions, you may add a little compass object to your game's UI which informs the player of every available exit from the current room. This way, you won't need to clutter your descriptions with mood-breaking lines explaining where the player can go to in practical terms, since there's a fixed spot on the screen where this information can be found.

10 · Inside Information (advanced)

What's a Library entry point?

More information in the DM: §A5

Inform provides around 20 opportunities for the game author to influence or control the Library's behaviour. You can do this from outside -- without having to change the standard header files -- because the Library provides 'hooks' on which, if you wish, you can hang suitable chunks of code. These hooks are actually calls to routines with pre-defined names; if your game provides a routine of that name, the Library calls it at the appropriate time. Your routine may either produce some supplementary output (for example, **DeathMessage()** provides an opportunity to explain how the player has 'died'), or return a value causing the Library to change its default behaviour (for example, **ChooseObjects()** allows you to influence how multiple or equivalent objects are selected).

Here's roughly how `parserm.h` invokes **ChooseObjects()** when deciding whether to include object 'j' in an ALL:

```
flag=1;
if (action_to_be == ##Take or ##Remove && parent(j)==actor) flag=0;
k=ChooseObjects(j,flag);
if (k==1) flag=1; else { if (k==2) flag=0; }
if (flag==1) {    ! Decided to INCLUDE it
...
} else {          ! Decided to EXCLUDE it
...
}
```

Now while there's no need to worry about the details of how that works, you will notice that the Library makes a perfectly normal call to **ChooseObjects()**, *without caring whether or not you've actually provided such a routine*. How can this work? normally, the compiler complains bitterly if you call a routine which doesn't exist. The answer is: there will always be a **ChooseObjects()** routine; if you don't provide one, the Library supplies a dummy version instead. Right at the end of `Grammar.h`, you'll find the line:

```
#Stub ChooseObjects   2;
```

More information in the DM: §38

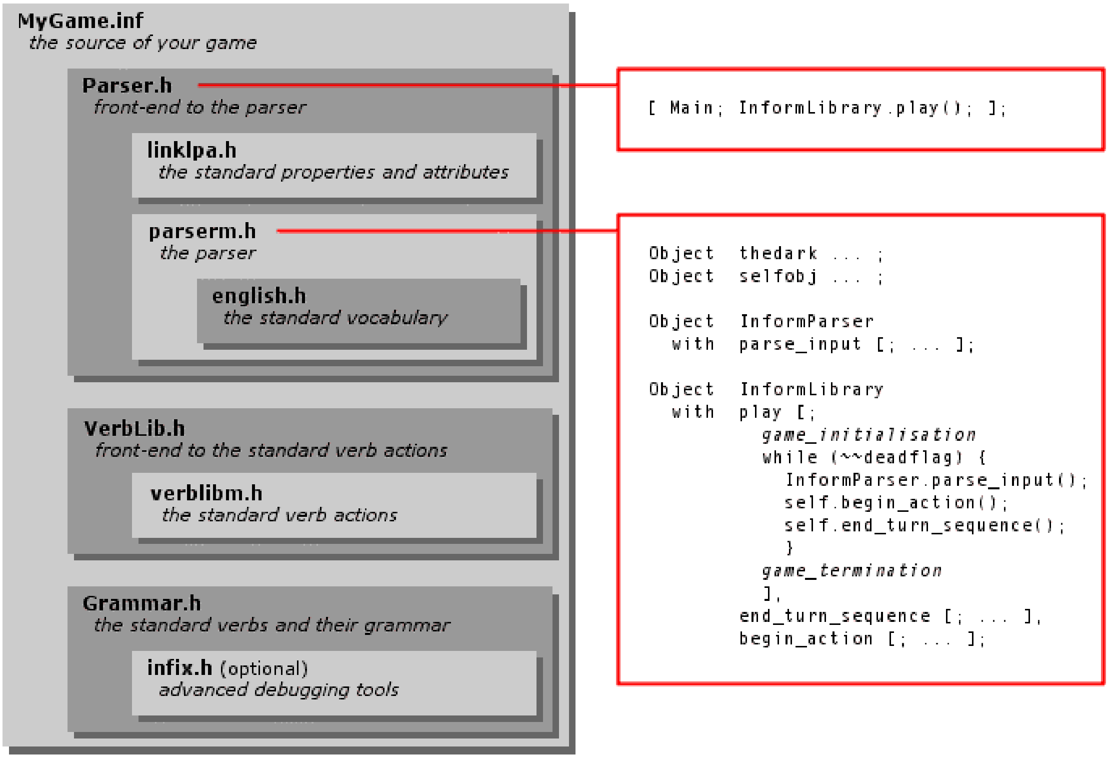
which is a shorthand way of saying:

```
#IfNDef ChooseObjects;
[ ChooseObjects a1 a2; rfalse; ];
#EndIf;
```

Two points are worth noting here: If you're providing an entry point routine, do so *before* the #Include of `Grammar.h`. And, that **rfalse** is important: for those entry point routines which return a value, **false** always triggers the Library's default behaviour.

Where are all those Library files used?

The Inform Library comprises nine header (`.h`) files; why so many, and where are they all used? You already know that every game must **#Include** the three top-most files `Parser.h`, `VerbLib.h` and `Grammar.h`; what is perhaps less familiar is the manner in which those three files in turn **#Include** the rest of the Library. The diagram illustrates how this works:



```
[ Main; InformLibrary.play(); ];
```

```
Object  thedark ... ;
Object  selfobj ... ;

Object  InformParser
  with  parse_input [; ... ];

Object  InformLibrary
  with  play [;
          game_initialisation
          while (~~deadflag) {
            InformParser.parse_input();
            self.begin_action();
            self.end_turn_sequence();
            }
          game_termination
          ],
        end_turn_sequence [; ... ],
        begin_action [; ... ];
```

The file `english.h` is actually included into `parserm.h` by the directive `Include "language__"`, where **language__** is a library variable (initialised to "english") which you can change with a compiler switch like `+language_name=french`. Also, if you count carefully, you'll notice that one file -- `linklv.h` -- isn't mentioned anywhere. That's because it applies only when using Modules, which as we all know is A Bad Idea.

The highlighted code to the right gives an overview of how the game runs (not that you need to know any of this other than from sheer curiosity). Every Inform program is required to have a **Main()** routine, but you don't write it yourself: the Library provides one in `Parser.h`. **Main()** simply invokes the **play** property of the **InformLibrary** object, and it's here that everything happens.

More information in the DM: §1.2 §4

The **play** property routine first sets up the game -- calls your **Initialise()** routine, moves **player** to **location**, prints the game banner, and so on -- and then loops until **deadflag** is non-zero. Each iteration of the loop represents one turn: parse a line typed by the player, perform the appropriate action, and then carry out standard end-of-turn processing. Finally, the player wins or dies, **deadflag** becomes non-zero, and the loop ends. The **play()** routine prints an appropriate message, and exits back to **Main()**. That in turn exits, and the game is over.

Why so many files? probably a combination of clean modularity, author flexibility and the requirements of the module linking scheme. Why are **parse_input()**, **play()**, **end_turn_sequence()** and **begin_action()** implemented as object properties rather than standalone routines? I haven't a clue.

Can I use Inform *without* the standard Library files?

Inform is actually a fairly general-purpose programming language, albeit with a few annoying limitations (for example, the Z-machine's Input/Output capabilities are somewhat limited, though Glulx is better in this respect). There's no reason why you shouldn't use the language for purposes other than writing Interactive Fiction and, if you do so, there's no particular need to **#Include** the library files `Parser.h`, `VerbLib.h` and `Grammar.h`. For example, this is a perfectly acceptable Inform program which runs on the Z-machine:

```
  [ Main;
      print "Hello world^";
  ];
```

That's pretty unexciting. Here's a slightly more useful Z-machine tool which scrambles text using ROT13: this involves replacing each letter with one that appears 13 characters later in the alphabet. (Note, by the way, that the Z-machine automatically converts all text to lowercase before storing it in the input buffer.)

More information in the DM: §1.2 §2.5

```
  ! ROT13 for the Z-machine

  Constant INPUTBUFFER_SIZE 100;
  Array    inputBuffer -> 2                  ! Max size, actual size,
                        + INPUTBUFFER_SIZE   ! then the characters,
                        + 1;               ! then a spare in case of overflow.

  [ Main i c;
      inputBuffer->0 = INPUTBUFFER_SIZE;
      while (true) {
          print "^^Enter some text> ";
          read inputBuffer 0;
          if (inputBuffer->1 == 0) quit;
          for (i=0 : i<inputBuffer->1 : i++) {
              c = inputBuffer->(i+2);
              switch (c) {
                'A' to 'M', 'a' to 'm': print (char) c + 13;
                'N' to 'Z', 'n' to 'z': print (char) c - 13;
                default:                print (char) c;
              }
          }
      }
  ];
```

Using Glulx, things are slightly more complex. For a start, the "Hello world" program won't quite run on the Glulx virtual machine which, unlike the Z-machine, doesn't automatically open a window in which to display the program's output. To make that happen, you need to include a few more lines:

More information on Andrew Plotkin's Glulx and Glk pages

```
  [ Main w;
      @setiosys 2 0;                   ! Select the Glk I/O system.
      w = glk($0023, 0, 0, 0, 3, 0);   ! Open a text-buffer window,
      glk($002F, w);             ! and select it as the current output stream.

      print "Hello world^";
  ];
```

The Z-machine has its Input/Output mechanisms built into the interpreter. Glulx is more generalised: it can use *any* suitable I/O system, though in practice that usually means Andrew Plotkin's **Glk**. So, **@setiosys 2 0;** specifies the use of Glk, the **glk($0023...)** call uses glk_window_open() to return the identifier of a newly-created window, and then the **glk($002F...)** call uses glk_set_window() to direct I/O to that window. Not that you need to worry about how it works: just copy those initialisation statements into your own program.

There's another problem to be overcome before our ROT13 tool works on Glulx: we can't use the Inform **read** statement. Instead, we need more glk() calls to fetch the line of input text:

```
  ! ROT13 for the Glulx VM

  Constant INPUTBUFFER_SIZE 100;
  Array    inputBuffer -> INPUTBUFFER_SIZE;

  Array    gg_event --> 4;
```

```
[ Main w i c;
    @setiosys 2 0;                    ! Select the Glk I/O system.
    w = glk($0023, 0, 0, 0, 3, 0);    ! Open a text-buffer window,
    glk($002F, w);             ! and select it as the current output stream.

    while (true) {
        print "^^Enter some text> ";
        glk($00D0, w, inputBuffer, INPUTBUFFER_SIZE, 0);
        while (true) {            ! Wait for RETURN to be pressed.
            glk($00C0, gg_event); ! LineInput is the only interesting event.
            if (gg_event-->0 == 3 && gg_event-->1 == w) break;
        }
        if (gg_event-->2 == 0) quit;
        for (i=0 : i<gg_event-->2 : i++) {
            c = inputBuffer->i;
            switch (c) {
              'A' to 'M', 'a' to 'm': print (char) c + 13;
              'N' to 'Z', 'n' to 'z': print (char) c - 13;
              default:                print (char) c;
            }
        }
    }

];
```

Here, the **glk($00D0...)** call uses glk_request_line_event() to request a line of text from the window, and then the **glk($00C0...)** call uses glk_select() to detect that the text has been typed. Again, don't get too hung up on the mechanics. In fact, if you #Include `infglk.h`, you can make things a little more readable, by using sensible names rather than mysterious numbers:

```
! ROT13 for the Glulx VM

Include "infglk";

Constant INPUTBUFFER_SIZE 100;
Array    inputBuffer -> INPUTBUFFER_SIZE;

Array    gg_event --> 4;

[ Main w i c;
    @setiosys 2 0;          ! Select the Glk I/O system.
    w = glk_window_open(0, 0, 0, wintype_TextBuffer, 0);
                            ! Open a text-buffer window,
    glk_set_window(w);      ! and select it as the current output stream.

    while (true) {
        print "^^Enter some text> ";
        glk_request_line_event(w, inputBuffer, INPUTBUFFER_SIZE, 0);
        while (true) {           ! Wait for RETURN to be pressed.
           glk_select(gg_event); ! LineInput is the only interesting event.
           if (gg_event-->0 == evtype_LineInput && gg_event-->1 == w) break;
        }
        if (gg_event-->2 == 0) quit;
        for (i=0 : i<gg_event-->2 : i++) {
            c = inputBuffer->i;
            switch (c) {
              'A' to 'M', 'a' to 'm': print (char) c + 13;
              'N' to 'Z', 'n' to 'z': print (char) c - 13;
              default:                print (char) c;
            }
        }
    }

];
```

What's all this stuff about message-passing?

More information in the DM: §3.9 §3.12

Inform games are constructed from four fundamental building blocks: Objects, Classes, Routines and Strings; other than primitive structures like variables and arrays, all of the components of a compiled game fit into one of these four categories. The most important category is the Object, and indeed it's the **InformLibrary** object, along with its more outgoing sister **InformParser**, which actually makes the game run.

How does it do that? Largely, by invoking the ***action*Sub()** associated with the player's current action, and by sending one-word messages to other objects -- the one(s) directly involved in that action, and others in the vicinity. The messages are the mechanism by which those objects (a) find out what's about to happen, and (b) have an opportunity to influence events. You can use the MESSAGES ON debugging command to watch them being sent.

Every message receives a reply; here's the syntax for sending a message and capturing the response:

```
  x = obj.message(p1,p2,...);
```

in which *obj* is the object to which the message is sent, *message* is the actual message, and *p1 p2* etc are optional parameters associated with the message. The reply is placed in variable *x*. For example, a command like DROP THE BLUE BALL is first parsed to give action=Drop and noun=blueball, and then the -- somewhat simplified -- flow of execution is:

```
  if (noun.before() == false) {
      move noun to parent(player);      ! These three statements
      if (noun.after() == false)        ! are the essence of
          print "Dropped.^";            ! the DropSub routine.
  }
```

The two messages involved in that action are highlighted; they look familar, don't they? You know them better in the guise of the **before** and **after** properties, and indeed that's exactly what they are: the **property** names possessed by an object are the **messages** which the object is prepared to accept, and the **values** of those properties determine what **replies** those messages will receive. You can increase the set of messages to which an object will respond simply by defining appropriate properties matching those message names.

You can also send messages to Classes, Routines and Strings, but here the rules a slightly different: the set of acceptable messages is pre-defined and can't be increased. This is a list of each possible message, the effect of sending it, and the reply that it will generate.

This message...	... has this effect...	... and returns this value
obj.*prop*() (when the value of *prop* is a routine)	Sets **self** to *obj*, **sender** to the sending object (often **InformLibrary**), and invokes the routine.	The value returned by the routine.
obj.*prop*() (when the value of *prop* is a string)	Outputs the string, followed by a newline.	**true**
obj.*prop*() (when the value of *prop* is **false**, **nothing**, zero or unspecified)	None.	**false**
obj.*prop*() (when *prop* is a common property not defined by *obj*)	None.	The default value of *prop*.

obj.*prop***()** (when the value of *prop* is a small number)	None.	The value of *prop*. Because 'small' is ill-defined, this usage requires care; a safer technique is simply `x=obj.prop;`
obj.*prop***()** (when the value of *prop* is a large or negative number)	Usually causes a run-time error.	
obj.*class*::*prop***()** (when the value of *prop* is any of the above)	Provided that object *obj* is of class *class*, sends the message to the *prop* property of the class, rather than to the (inherited or over-ridden) *prop* property of the object.	The value from that same *class* property.
class.**remaining()**	None.	The number of pre-defined object instances of *class* which are not currently active.

class.**create()**
or
class.**create(***p1*,*p2*,*p3***)**
	Creates an object *obj* of this class (actually, activates a pre-defined instance) with a parent of **nothing**. If *obj* has inherited a **create** property routine from *class*, sends a further message *obj*.**create()** or *obj*.**create(***p1*,*p2*,*p3***)**.	*obj*, or **nothing** if no more instances can be activated.

class.**recreate(***obj***)**
or
class.**recreate(***obj*,*p1*,*p2*,*p3***)**
	Re-initialises *obj* to its state at creation. If *obj* has inherited a **create** property routine from *class*, sends a further message *obj*.**create()** or *obj*.**create(***p1*,*p2*,*p3***)**.	**false**

class.**destroy(***obj***)**	Destroys an object of this class actually, de-activates a pre-defined instance). If *obj* has inherited a **destroy** property routine from *class*, sends a further message *obj*.**destroy()**.	**false**
class.**copy(***obj*,*source_obj***)**	Sets the property and attribute values of *obj* equal to those of *source_obj*. Both objects must be members of *class*.	**false**
routine.**call(***p1*,*p2*,...**)**	Invokes the routine; essentially equivalent to: *routine***(***p1*,*p2*,...**)**	The value returned by *routine*.
string.**print()**	Outputs the string, followed by a newline; equivalent to: **print_ret "***string***"**	**true**

string.**print_to_array(***array***)**	Writes the number of characters in the string into word *array*-->0, and the individual string characters into bytes *array*->2, *array*->3, *array*->4, etc; equivalent to:
	@input_stream 3 *array***;**
	print "*string***";**
	@input_stream -3;
		The number of characters in *string*.

What actually happens at the start and end of each turn?

Start: The **parse_input** property of the **InformParser** object is responsible for:

1. displaying the prompt, by calling **L__M(##Prompt)**,
2. running the optional **AfterPrompt** entry point routine,
3. refreshing the status line, by calling **DrawStatusLine()**,
4. reading the player's typed input,
5. dealing with an AGAIN, OOPS or UNDO,
6. parsing the player's input.

More information in the DM: §22

End: The **end_turn_sequence** property of the **InformLibrary** object is responsible for:

More information in the DM: §20

1. adjusting the **turns** and **the_time** variables,
2. running all active daemons and timers,
3. running any **each_turn** property for the **location**, and then for each object in scope,
4. running the optional **TimePasses** entry point routine,
5. checking whether the location has light,

giving the **moved** attribute, and optionally awarding scores, to any objects which the player has just picked up.

What's the difference between a Daemon and a Timer?

Not a great deal: each is a routine, defined as an object's embedded property routine, which has no effect until you explicitly set it running. The difference lies in what happens next: a daemon then runs a **daemon()** routine automatically at the end of each turn (until you stop it or the game finishes); a timer does nothing for a specified number of turns (stored in **time_left**), and then runs a **time_out()** routine once only. To illustrate the distinction, consider these two simple examples. The blue box defines a daemon which makes the box 'tick' for as long as it's switched on, while the red box defines a timer which makes the box explode without notice three turns after being switched on.

More information in the DM: §20

```
Object  -> "blue box"
  with  name 'blue' 'box',
        after [;
            SwitchOn: StartDaemon(self);
            SwitchOff: StopDaemon(self);
            ],
        daemon [;
            if (IndirectlyContains(location,self))
                "^An ominous ticking noise is coming from the box.";
            ],
  has   switchable ~on;

Object  -> "red box"
  with  name 'red' 'box',
        after [;
            SwitchOn: StartTimer(self,3);
            SwitchOff: StopTimer(self);
            ],
        time_out [;
            if (IndirectlyContains(location,self))
                print "^The box explodes into a zillion fragments.^";
            remove self;
            ],
        time_left 0,
  has   switchable ~on;
```

The blue daemon runs once each turn from the turn where you call **StartDaemon()** until the turn where you call **StopDaemon()**; the red timer waits three turns after you call **StartTimer()**, and then runs explosively... unless the player switches off the red box in time. To allow for the player switching on a box and then wandering away, both daemon and timer print their output only if they're in the same room as the player.

In fact, because a daemon runs at the end of every turn, you can make it do the work of a timer as well. Here the blue and red boxes have been combined:

```
Object  -> "purple box"
  with  name 'purple' 'box',
        after [;
            SwitchOn: self.time_left = 3; StartDaemon(self);
            SwitchOff: StopDaemon(self);
            ],
        daemon [;
            if (self.time_left > 0) {
                (self.time_left)--;
                if (IndirectlyContains(location,self))
                    "^An ominous ticking noise is coming from the box.";
                }
            else {
                if (IndirectlyContains(location,self))
                    print "^The box explodes into a zillion fragments.^";
                remove self;
                }
            ],
        time_left 0,
  has   switchable ~on;
```

In a complicated game many daemons can be active at the same time, and you may then encounter problems with the order in which they run (which is not under your direct control); this is most evident with daemons which are frequently stopped and started. For example, suppose that one daemon causes a nasty dwarf to roam though the rooms in your game, while another causes the player's sword to glow when evil is nearby. You really need to ensure that the 'dwarf' daemon runs first, so that the 'sword' daemon can immediately react if the dwarf moves into the player's location. There are two standard techniques for controlling the order in which daemons run. One method is to foresake the Library's standard **StartDaemon()** and **StopDaemon()** routines, and instead run continuously a single 'superdaemon' which in turn calls the real daemons, in the desired order, *if* a suitable property (or attribute) is set. For example, to ensure that three daemons run, if at all, in the order X-Y-Z, you could write:

```
[ Initialise;
        StartDaemon(superdaemon);
        ...
        ];

Object  superdaemon
  with  daemon [;
            if (objX.daemon_is_active) objX.daemon();
            if (objY.daemon_is_active) objY.daemon();
            if (objZ.daemon_is_active) objZ.daemon();
            ];

Object  objX
  with  daemon [; ... ],
        daemon_is_active false;

Object  objY
  with  daemon [; ... ],
        daemon_is_active true;

Object  objZ
  with  daemon [; ... ],
        daemon_is_active true;
```

In this example, you'd code the assignment `objX.daemon_is_active = true;` instead of the routine call `StartDaemon(objX);` and `objX.daemon_is_active = false;` instead of `StopDaemon(objX);`.

The other approach is to use Andrew Plotkin's `daemons.h` from the Archive, which enables you to assign priority levels to your daemons (and timers). Higher priorities always run before lower ones, so you could then write:

```
Object  objX
  with  daemon [; ... ],
        daemon_priority 50;

Object  objY
  with  daemon [; ... ],
        daemon_priority 30;

Object  objZ
  with  daemon [; ... ],
        daemon_priority 10;
```

No superdaemon is needed, and this time the standard **StartDaemon()** and **StopDaemon()** routines can be used as normal.

If you're not concerned about daemon/timer sequence, you can ignore both of these complications and just follow the basic procedures. Otherwise, which one you adopt is down to you: the superdaemon approach avoids the need to patch Library routines, but doesn't control timers (which run in unspecified order, always *after* the daemons). The priority system controls both daemons *and* timers, but doesn't allow one daemon/timer to start another (though stopping isn't a problem).

What's the difference between a Daemon and an each_turn property?

Again, each is a routine, defined as an object's embedded property routine. A **daemon** property needs to be explicitly started, whereupon it runs at the end of each turn until stopped. An **each_turn** property doesn't need to be started; it is automatically run by the parser, but only at the end of those turns when the associated object is in scope.

Typically, use a daemon to control a process which continues regardless of whether the player is there to watch it, or not (for example, the exploding box). Use an each_turn to control a process which might as well be suspended if there's no audience (for example, telling the player about the sleeping giant who grunts and snores in the corner of the cave).

Why don't my daemons run at the start of a game?

One fairly frequent use for a **daemon** or **each_turn** property is to add atmospheric information. For example, consider this specialised Room class:

```
Class   Coastal
  with  each_turn "^The sound and smell of the sea is never far away.",
  has   light;
```

which simply prints the string at the end of the description for each Coastal room. Well, nearly always; this *doesn't* happen at the very start of a game:

```
On the beach
An interactive Shute story.
Release 1 / Serial number 040112 / Inform v6.30 Library 6/11 S
```

```
By the sea
You stand on a small dune overlooking an ocean inlet, watching the waves
break gently on the sandy shore. A path leads inland to the east.

>
```

Since the game has just displayed the description of a Coastal room, you might expect to see **each_turn** adding its small contribution. The reason that you don't see it is because timers, daemons and each_turn properties are triggered only at the *end* of a turn, just after **turns** and **the_time** have been incremented (see the topic above). The first turn doesn't end until the player types something and presses Return.

Although this library behaviour is logical, it can sometimes be a nuisance, so here's how to work round it: by defining a **LookRoutine()** entry point. A **LookRoutine()** is called by the library at the end of every Look action, including the one which prints the very first description. So, by detecting that situation -- that it's the start of a game -- you can yourself activate timers and daemons, each_turn properties or both. Here's the code to run just the current location's **each_turn** property, which is all that we need to crack the problem illustrated above:

```
[ LookRoutine;
    if (turns == 1 && verb_word == 0)
        location.each_turn();
];
```

(The check on **verb_word** catches the player's first command being another LOOK.) If you want to run *all* appropriate each_turns, or trigger the timers and daemons, then it's slightly trickier; you need to copy a block of code from **InformLibrary.end_turn_sequence**, defined two-thirds of the way through parserm.h:

```
[ LookRoutine;
    if (turns == 1 && verb_word == 0) {
        ! run all timers and daemons
        for (i=0 : i<active_timers : i++) {
            if (deadflag) return;
            j = the_timers-->i;
            if (j ~= 0) {
                if (j & WORD_HIGHBIT) RunRoutines(j&~WORD_HIGHBIT, daemon);
                else {
                    if (j.time_left == 0) {
                        StopTimer(j);
                        RunRoutines(j, time_out);
                    }
                    else
                        j.time_left = j.time_left-1;
                }
            }
        }
        ! run each_turn for all in-scope objects
        scope_reason = EACH_TURN_REASON; verb_word = 0;
        DoScopeAction(location);
        SearchScope(ScopeCeiling(player), player, 0);
        scope_reason = PARSING_REASON;
    }
];
```

Fortunately, version 6/11 of the Library makes this much simpler:

```
[ LookRoutine;
    if (turns == START_MOVE && verb_word == 0) {
        RunTimersAndDaemons();
        RunEachTurnProperties();
    }
];
```

How do I compile a game as Version 3?

Occasionally, the only interpreter available for your computer doesn't support Z-code higher than Version 3 (standard). To compile for this, use:

- the **6.15 Compiler** from ...i--/c--/inform6/executables with the **-v3** switch, and
- the **6/2 Library** from ...i--/c--/inform6/library/old

Can I combine a game and an interpreter in a single file?

Very occasionally, it's convenient to distribute a single ready-to-run package containing both a Z-code game file *and* a suitable interpreter. I know of a couple of tools which do just this:

- John Holder's Jzip interpreter includes the JZEXE bundling tool (PC and UNIX)
- L Ross Raszewski has a more generalized utility called BundleMonkey (PC only)

Try it: here's my tiny (50KB) Cloak of Darkness example, in the form of a Jzip bundle (155KB), a Monkey bundle with WinFrotz (275KB), and a plain WinZip archive with WinFrotz (95KB).

Could you explain how character sets are handled?

More information in:
the DM (§1.11, §36 and Table 2)
the Technical Manual (§8.3 and §12.3)
the Z-Machine Standards 1.0 document (§3)
the FAQ topic on Writing a French game

The Z-machine's character handling is one of those topics which is simple until you start thinking about it, whereupon it takes a turn for the trickier (or maybe it's just me). Although most of the necessary information is available, it's scattered across several locations; this is an attempt to pull together the basics in one place. Bear in mind that most of what follows is specific to the Z-machine; Glulx handles characters quite differently (and currently doesn't offer any support for extended character sets). We'll set the scene with a trip down memory lane.

Character encoding

Historically, most computers have stored arbitrary character sequences -- 'strings' -- with one character per byte of storage. An eight-bit byte supports 256 different character codes, more than enough for the letters, digits, punctuation marks and control codes which were common currency among the English-speaking fathers of modern-day computing. Fairly early on, the obvious advantages of standardisation resulted in the American Standard Code for Information Interchange (ASCII), defined in 1968 as ANSI X3.4, and later adopted as ISO 646. This encoding allocated characters in a reasonably logical manner to the first 128 possible values (hex $00...$7F) -- so that for example $21 is an exclamation mark, $31 is the digit '**1**', $41 is the letter '**A**' and $61 is the letter '**a**' -- and is still almost universally used today (only IBM's EBCDIC, oriented towards mainframe punched cards, survives as an alternative).

ASCII provides exactly 26 letter forms in upper and lower case, sufficient only to encode text in English, Hawaiian, Latin and Swahili. As the need to handle other languages grew, a wide and incompatible range of alternative allocations were devised for the infrequently-used standard characters '**#$@[\]^`{|}~**', and for the remaining 128 values (hex $80...$FF) which ASCII didn't specify.

Around the mid-1980s, in an attempt to escape total chaos, ISO 8859 defined a series of nine ways of encoding all 256 byte values. Values $00...$7F (the ASCII encoding) and $80…$9F (additional control codes) are identical in each of the nine encodings; the differences lie in the 96 characters with values $A0...$FF. For example, ISO 8859-1 allocates those values to characters commonly used in Western European text, ISO 8859-4 is appropriate for North European text, ISO 8859-7 is for Greek text, and so on.

Although ISO 8859 was a step in the right direction, it still left an awful lot of languages out in the cold; basically, it just isn't possible to squeeze all of those incompatible character sets into an eight-bit encoding scheme. As so, in 1991 we get to ISO 10646 -- Unicode -- which uses a sixteen-bit encoding (with a twenty-bit extension zone for when that runs out). Sixteen bits support 65,536 different character codes, of which roughly 50,000 have currently been allocated. Conveniently, the first 256 Unicode characters are the same as ISO 8859-1, so that $0041 and $0061 are still '**A**' and '**a**', while $00C1 and $00E1 remain consistently '**Á**' and '**á**'.

More information can be found in the excellent sites of Roman Czyborra (offline?) and Alan Wood, and at unicode.org

> TECHNICAL NOTE: The rest of this page assumes that you're using version 6.30 (or higher) of the Inform compiler. There were significant problems with character set handling in earlier versions of the compiler, and I strongly advise you to upgrade before experimenting with these techniques.

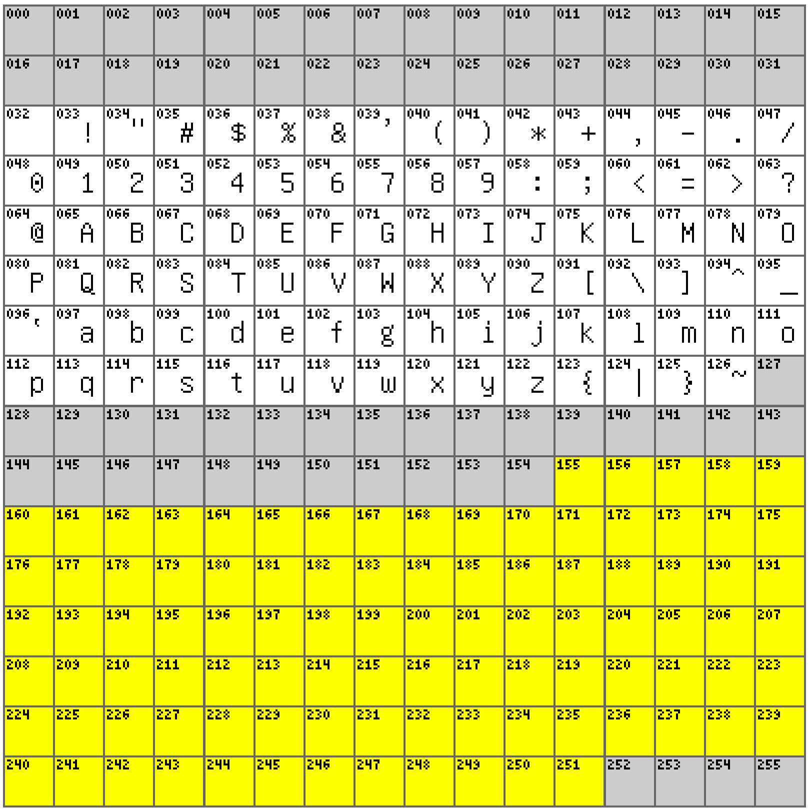
The ZSCII character set

Character strings in the Z-machine use the ZSCII encoding ('Z', unsurprisingly, stands for 'Zork'). ZSCII is an eight-bit character set, leading to a repertoire of 256 character codes.

> TECHNICAL NOTE: The Z-Machine Standard 1.1 Proposal includes a mechanism to extend this, offering direct access to Unicode strings.

In the ZSCII character set, some of the 256 values are reserved for control purposes and others are unused; these values are shown in grey. The values in the range 32...126 -- shown in white -- are standard encodings which cannot be changed (and are the same as the ASCII / ISO 8859 / Unicode characters in that range). The values in the range 155...251 -- shown in yellow -- are 'extra characters' which you can allocate. There are two ways of populating this ZSCII range with characters of your choice:

- using the **C*n*** compiler switch;
- using the **Zcharacter** directive.

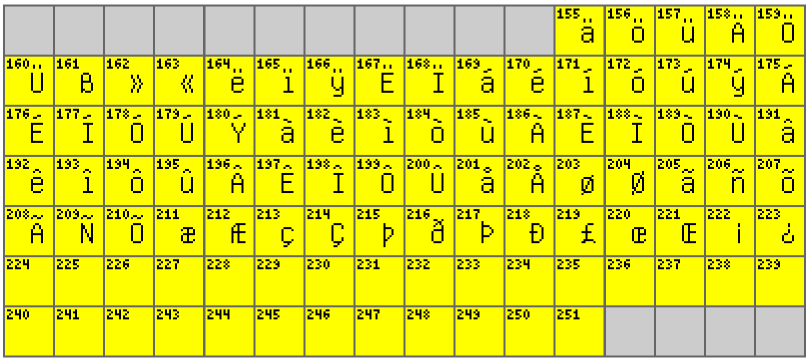


000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
032	033 !	034 "	035 #	036 $	037 %	038 &	039 '	040 (	041)	042 *	043 +	044 ,	045 -	046 .	047 /
048 0	049 1	050 2	051 3	052 4	053 5	054 6	055 7	056 8	057 9	058 :	059 ;	060 <	061 =	062 >	063 ?
064 @	065 A	066 B	067 C	068 D	069 E	070 F	071 G	072 H	073 I	074 J	075 K	076 L	077 M	078 N	079 O
080 P	081 Q	082 R	083 S	084 T	085 U	086 V	087 W	088 X	089 Y	090 Z	091 [	092 \	093]	094 ^	095 _
096 `	097 a	098 b	099 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Allocating extra characters using the C switch

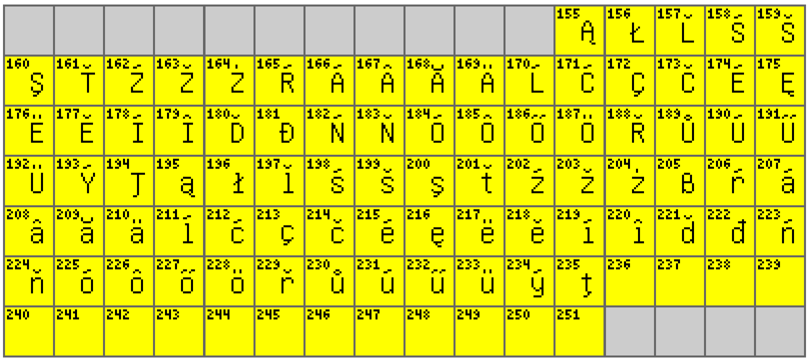
At compile-time, you can choose to populate the yellow range with characters taken from one of the nine ISO 8859 series. You can set the switch either on the command line (**-C*n***) or as a special comment at the very start of your source file (**!% -C*n***).

If you specify **C0** or **C1**, or if you omit the **C** switch altogether, then the compiler takes letters and symbols from ISO 8859-1. Not all characters are taken, and their ZSCII order is not the same as in ISO 8859-1. The result is Infocom's version of ZSCII, appropriate for Western European games: English, French, German, Spanish, Swedish, etc. This is what happens by default.



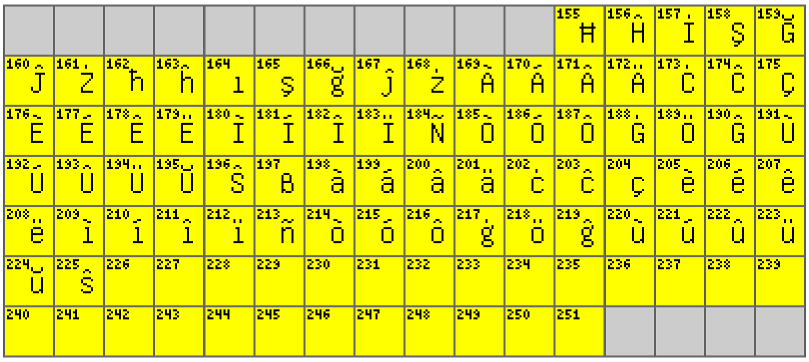
											155 ä	156 ö	157 ü	158 Ä	159 Ö
160 Ü	161 ß	162 »	163 «	164 ë	165 ï	166 ÿ	167 Ë	168 Ï	169 á	170 é	171 í	172 ó	173 ú	174 ý	175 Á
176 É	177 Í	178 Ó	179 Ú	180 Ý	181 à	182 è	183 ì	184 ò	185 ù	186 À	187 È	188 Ì	189 Ò	190 Ù	191 â
192 ê	193 î	194 ô	195 û	196 Â	197 Ê	198 Î	199 Ô	200 Û	201 å	202 Å	203 ø	204 Ø	205 ã	206 ñ	207 õ
208 Ã	209 Ñ	210 Õ	211 æ	212 Æ	213 ç	214 Ç	215 þ	216 ð	217 Þ	218 Ð	219 £	220 œ	221 Œ	222 ¡	223 ¿
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251				

If you specify **C2**, the compiler takes all of the letter characters, but not in order, from ISO 8859-2. The result is a version of ZSCII appropriate for Eastern European games: Croatian, Hungarian, Polish, etc.



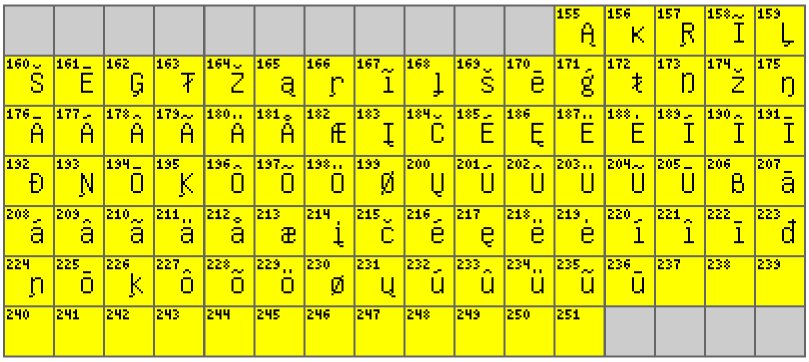
											155 Ą	156 Ł	157 Ľ	158 Ś	159 Š
160 Ş	161 Ť	162 Ź	163 Ž	164 Ż	165 Ŕ	166 Á	167 Â	168 Ă	169 Ä	170 Ĺ	171 Ć	172 Ç	173 Č	174 É	175 Ę
176 Ë	177 Ě	178 Í	179 Î	180 Ď	181 Đ	182 Ń	183 Ň	184 Ó	185 Ô	186 Ő	187 Ö	188 Ř	189 Ů	190 Ú	191 Ű
192 Ü	193 Ý	194 Ţ	195 ą	196 ł	197 ľ	198 ś	199 š	200 ş	201 ť	202 ź	203 ž	204 ż	205 ß	206 ŕ	207 á
208 â	209 ă	210 ä	211 ĺ	212 ć	213 ç	214 č	215 é	216 ę	217 ë	218 ě	219 í	220 î	221 ď	222 đ	223 ń
224 ň	225 ó	226 ô	227 ő	228 ö	229 ř	230 ů	231 ú	232 ű	233 ü	234 ý	235 ţ	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251				

If you specify **C3**, the compiler takes all of the letter characters, in order, from ISO 8859-3. The result is a version of ZSCII appropriate for Southern European games: Maltese, also Esperanto.



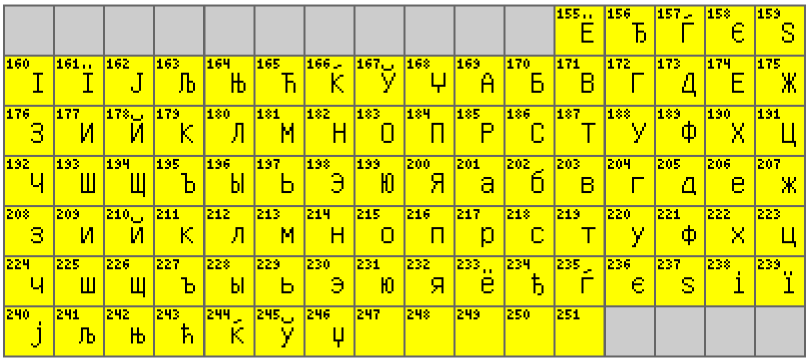
											155 Ħ	156 Ĥ	157 İ	158 Ş	159 Ğ
160 Ĵ	161 Ż	162 ħ	163 ĥ	164 ı	165 ş	166 ğ	167 ĵ	168 ż	169 À	170 Á	171 Â	172 Ä	173 Ċ	174 Ĉ	175 Ç
176 È	177 É	178 Ê	179 Ë	180 Ì	181 Í	182 Î	183 Ï	184 Ñ	185 Ò	186 Ó	187 Ô	188 Ġ	189 Ö	190 Ĝ	191 Ù
192 Ú	193 Û	194 Ü	195 Ŭ	196 Ŝ	197 ß	198 à	199 á	200 â	201 ä	202 ċ	203 ĉ	204 ç	205 è	206 é	207 ê
208 ë	209 ì	210 í	211 î	212 ï	213 ñ	214 ò	215 ó	216 ô	217 ġ	218 ö	219 ĝ	220 ù	221 ú	222 û	223 ü
224 ŭ	225 ŝ	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251				

If you specify **C4**, the compiler takes all of the letter characters, in order, from ISO 8859-4. The result is a version of ZSCII appropriate for Northern European games: Estonian, Latvian, Lithuanian, etc.



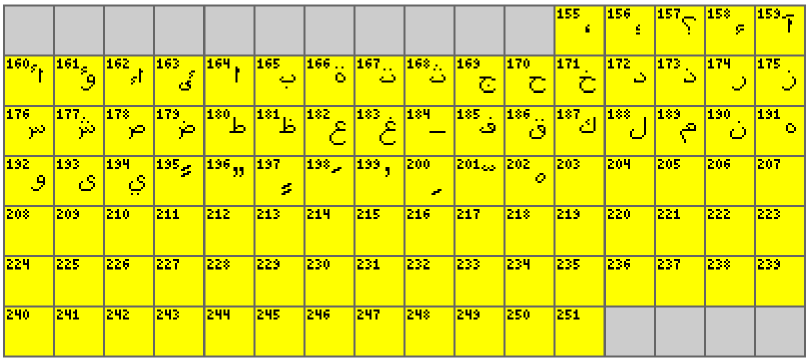
											155 Ą	156 ĸ	157 Ŗ	158 Ĩ	159 Ļ
160 Š	161 Ē	162 Ģ	163 Ŧ	164 Ž	165 ą	166 ŗ	167 ĩ	168 ļ	169 š	170 ē	171 ģ	172 ŧ	173 Ŋ	174 ž	175 ŋ
176 Ā	177 Á	178 Â	179 Ã	180 Ä	181 Å	182 Æ	183 Į	184 Č	185 É	186 Ę	187 Ë	188 Ė	189 Í	190 Î	191 Ī
192 Đ	193 Ņ	194 Ō	195 Ķ	196 Ô	197 Õ	198 Ö	199 Ø	200 Ų	201 Ú	202 Û	203 Ü	204 Ũ	205 Ū	206 ß	207 ā
208 á	209 â	210 ã	211 ä	212 å	213 æ	214 į	215 č	216 é	217 ę	218 ë	219 ė	220 í	221 î	222 ī	223 đ
224 ņ	225 ō	226 ķ	227 ô	228 õ	229 ö	230 ø	231 ų	232 ú	233 û	234 ü	235 ũ	236 ū	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251				

If you specify **C5**, the compiler takes all of the letter characters, in order, from ISO 8859-5. The result is a version of ZSCII appropriate for Cyrillic games: Bulgarian, Russian, Serbian, etc.



											155 Ё	156 Ђ	157 Ѓ	158 Є	159 Ѕ
160 І	161 Ї	162 Ј	163 Љ	164 Њ	165 Ћ	166 Ќ	167 Ў	168 Џ	169 А	170 Б	171 В	172 Г	173 Д	174 Е	175 Ж
176 З	177 И	178 Й	179 К	180 Л	181 М	182 Н	183 О	184 П	185 Р	186 С	187 Т	188 У	189 Ф	190 Х	191 Ц
192 Ч	193 Ш	194 Щ	195 Ъ	196 Ы	197 Ь	198 Э	199 Ю	200 Я	201 а	202 б	203 в	204 г	205 д	206 е	207 ж
208 з	209 и	210 й	211 к	212 л	213 м	214 н	215 о	216 п	217 р	218 с	219 т	220 у	221 ф	222 х	223 ц
224 ч	225 ш	226 щ	227 ъ	228 ы	229 ь	230 э	231 ю	232 я	233 ё	234 ђ	235 ѓ	236 є	237 ѕ	238 і	239 ї
240 ј	241 љ	242 њ	243 ћ	244 ќ	245 ў	246 џ	247	248	249	250	251				

If you specify **C6**, the compiler takes all of the letter characters, in order, from ISO 8859-6. The result is a version of ZSCII appropriate for Arabic games.



											155 ،	156 ؛	157 ؟	158 ء	159 آ
160 أ	161 ؤ	162 إ	163 ئ	164 ا	165 ب	166 ة	167 ت	168 ث	169 ج	170 ح	171 خ	172 د	173 ذ	174 ر	175 ز
176 س	177 ش	178 ص	179 ض	180 ط	181 ظ	182 ع	183 غ	184 ـ	185 ف	186 ق	187 ك	188 ل	189 م	190 ن	191 ه
192 و	193 ى	194 ي	195 ً	196 ٌ	197 ٍ	198 َ	199 ُ	200 ِ	201 ّ	202 ْ	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251				

If you specify **C7**, the compiler takes all of the letter characters, in order, from ISO 8859-7. The result is a version of ZSCII appropriate for Greek games.

											155 ΄	156 ΅	157 Ά	158 Έ	159 Ή
160 Ί	161 Ό	162 Ύ	163 Ώ	164 ΐ	165 Α	166 Β	167 Γ	168 Δ	169 Ε	170 Ζ	171 Η	172 Θ	173 Ι	174 Κ	175 Λ
176 Μ	177 Ν	178 Ξ	179 Ο	180 Π	181 Ρ	182 Σ	183 Τ	184 Υ	185 Φ	186 Χ	187 Ψ	188 Ω	189 Ϊ	190 Ϋ	191 ά
192 έ	193 ή	194 ί	195 ΰ	196 α	197 β	198 γ	199 δ	200 ε	201 ζ	202 η	203 θ	204 ι	205 κ	206 λ	207 μ
208 ν	209 ξ	210 ο	211 π	212 ρ	213 ς	214 σ	215 τ	216 υ	217 φ	218 χ	219 ψ	220 ω	221 ϊ	222 ϋ	223 ό
224 ύ	225 ώ	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251				

If you specify **C8**, the compiler takes all of the letter characters, in order, from ISO 8859-8. The result is a version of ZSCII appropriate for Hebrew games.

											155 א	156 ב	157 ג	158 ד	159 ה
160 ו	161 ז	162 ח	163 ט	164 י	165 ך	166 כ	167 ל	168 ם	169 מ	170 ן	171 נ	172 ס	173 ע	174 ף	175 פ
176 ץ	177 צ	178 ק	179 ר	180 ש	181 ת	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251				

If you specify **C9**, the compiler takes all of the letter characters, in order, from ISO 8859-9. The result is a version of ZSCII appropriate for Turkish games.

											155 À	156 Á	157 Â	158 Ã	159 Ä
160 Å	161 Æ	162 Ç	163 È	164 É	165 Ê	166 Ë	167 Ì	168 Í	169 Î	170 Ï	171 Ğ	172 Ñ	173 Ò	174 Ó	175 Ô
176 Õ	177 Ö	178 Ø	179 Ù	180 Ú	181 Û	182 Ü	183 İ	184 Ş	185 ß	186 à	187 á	188 â	189 ã	190 ä	191 å
192 æ	193 ç	194 è	195 é	196 ê	197 ë	198 ì	199 í	200 î	201 ï	202 ğ	203 ñ	204 ò	205 ó	206 ô	207 õ
208 ö	209 ø	210 ù	211 ú	212 û	213 ü	214 ı	215 ş	216 ÿ	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251				

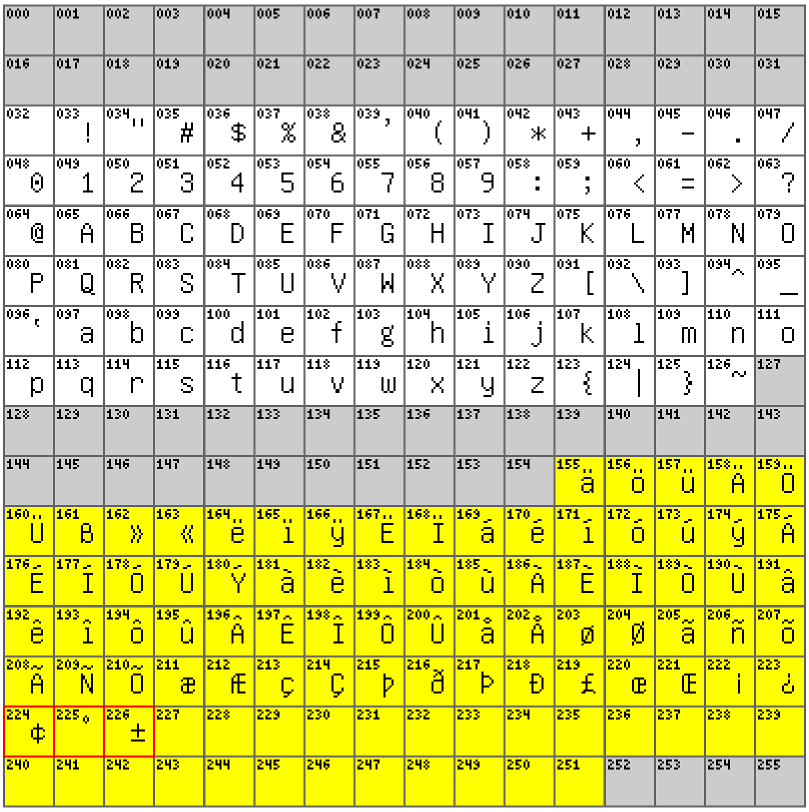
Note that in all nine cases, some of the yellow range remains free. You can allocate additional characters here, as explained next.

Allocating extra characters using the Zcharacter directive

Having first populated the yellow range with a selection of ISO 8859 characters, you can then choose either to supplement those characters with others of your own choosing, or to replace them completely by up to 96 characters which you specify within the source file.

For example, you could start with the default allocation, and then add the three characters '**¢°±**', by using this **Zcharacter** directive at the very start of your source file:

```
  Zcharacter table + '@{00A2}' '@{00B0}' '@{00B1}';
```

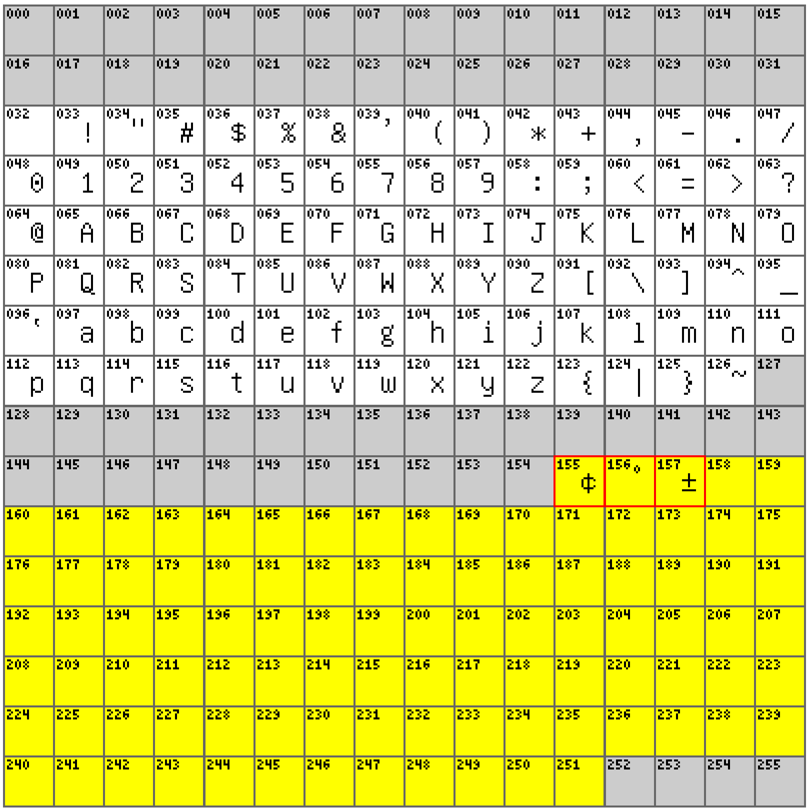


000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
032	033 !	034 "	035 #	036 $	037 %	038 &	039 '	040 (	041)	042 *	043 +	044 ,	045 -	046 .	047 /
048 0	049 1	050 2	051 3	052 4	053 5	054 6	055 7	056 8	057 9	058 :	059 ;	060 <	061 =	062 >	063 ?
064 @	065 A	066 B	067 C	068 D	069 E	070 F	071 G	072 H	073 I	074 J	075 K	076 L	077 M	078 N	079 O
080 P	081 Q	082 R	083 S	084 T	085 U	086 V	087 W	088 X	089 Y	090 Z	091 [	092 \	093]	094 ^	095 _
096 `	097 a	098 b	099 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155 ä	156 ö	157 ü	158 Ä	159 Ö
160 Ü	161 ß	162 »	163 «	164 ë	165 ï	166 ÿ	167 Ë	168 Ï	169 á	170 é	171 í	172 ó	173 ú	174 ý	175 Á
176 É	177 Í	178 Ó	179 Ú	180 Ý	181 à	182 è	183 ì	184 ò	185 ù	186 À	187 È	188 Ì	189 Ò	190 Ù	191 â
192 ê	193 î	194 ô	195 û	196 Â	197 Ê	198 Î	199 Ô	200 Û	201 å	202 Å	203 ø	204 Ø	205 ã	206 ñ	207 õ
208 Ã	209 Ñ	210 Õ	211 æ	212 Æ	213 ç	214 Ç	215 þ	216 ð	217 Þ	218 Ð	219 £	220 œ	221 Œ	222 ¡	223 ¿
224 ¢	225 °	226 ±	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Alternatively, you could replace the initial allocation by just those three characters, using this (very similar) **Zcharacter** directive:

```
  Zcharacter table '@{00A2}' '@{00B0}' '@{00B1}';
```

Both these **Zcharacter** directives expect a list of character values, each given in single quotes. The construct **@{00A2}** specifies a Unicode character value in hexadecimal -- 00A2 is the value for the cent sign '**¢**', 00B0 is the degree sign '**°**', and 00B1 is the plus-minus sign '**±**'.



000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
032	033 !	034 "	035 #	036 $	037 %	038 &	039 '	040 (	041)	042 *	043 +	044 ,	045 -	046 .	047 /
048 0	049 1	050 2	051 3	052 4	053 5	054 6	055 7	056 8	057 9	058 :	059 ;	060 <	061 =	062 >	063 ?
064 @	065 A	066 B	067 C	068 D	069 E	070 F	071 G	072 H	073 I	074 J	075 K	076 L	077 M	078 N	079 O
080 P	081 Q	082 R	083 S	084 T	085 U	086 V	087 W	088 X	089 Y	090 Z	091 [	092 \	093]	094 ^	095 _
096 `	097 a	098 b	099 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155 ¢	156 °	157 ±	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

TECHNICAL NOTE: Every Z-code interpreter requires a table to convert ZSCII values back into printable Unicode characters. A default table -- matching the default ZSCII character set -- is built into the interpreter. However, if you supply a switch **C2**...**C9**, or if you use either of these **Zcharacter** directives, then the default table becomes inadequate, and the compiler must include a replacement table within the game file. This is the 'Unicode translation table', whose address is given in the Z-machine Header Extension area (whose address is itself given in the game's header at address HDR_EXTENSION-->0). You can display the table by including Roger Firth's `dump.h` library extension, and typing DUMP UCODE.

Just for completeness, here are **Zcharacter** directives which set up tables equivalent to those created by the switches **C1**-**C9**:

```
Zcharacter table        ! C0/C1 - Latin 1 (West European)

    '@{00E4}' '@{00F6}' '@{00FC}' '@{00C4}' '@{00D6}' '@{00DC}' '@{00DF}' '@{00BB}'
    '@{00AB}' '@{00EB}' '@{00EF}' '@{00FF}' '@{00CB}' '@{00CF}' '@{00E1}' '@{00E9}'
    '@{00ED}' '@{00F3}' '@{00FA}' '@{00FD}' '@{00C1}' '@{00C9}' '@{00CD}' '@{00D3}'
    '@{00DA}' '@{00DD}' '@{00E0}' '@{00E8}' '@{00EC}' '@{00F2}' '@{00F9}' '@{00C0}'
    '@{00C8}' '@{00CC}' '@{00D2}' '@{00D9}' '@{00E2}' '@{00EA}' '@{00EE}' '@{00F4}'
    '@{00FB}' '@{00C2}' '@{00CA}' '@{00CE}' '@{00D4}' '@{00DB}' '@{00E5}' '@{00C5}'
    '@{00F8}' '@{00D8}' '@{00E3}' '@{00F1}' '@{00F5}' '@{00C3}' '@{00D1}' '@{00D5}'
    '@{00E6}' '@{00C6}' '@{00E7}' '@{00C7}' '@{00FE}' '@{00F0}' '@{00DE}' '@{00D0}'
    '@{00A3}' '@{0153}' '@{0152}' '@{00A1}' '@{00BF}';
```

```
Zcharacter table        ! C2 - Latin 2 (East European)

    '@{0104}' '@{0141}' '@{013D}' '@{015A}' '@{0160}' '@{015E}' '@{0164}' '@{0179}'
    '@{017D}' '@{017B}' '@{0154}' '@{00C1}' '@{00C2}' '@{0102}' '@{00C4}' '@{0139}'
    '@{0106}' '@{00C7}' '@{010C}' '@{00C9}' '@{0118}' '@{00CB}' '@{011A}' '@{00CD}'
    '@{00CE}' '@{010E}' '@{0110}' '@{0143}' '@{0147}' '@{00D3}' '@{00D4}' '@{0150}'
    '@{00D6}' '@{0158}' '@{016E}' '@{00DA}' '@{0170}' '@{00DC}' '@{00DD}' '@{0162}'
    '@{0105}' '@{0142}' '@{013E}' '@{015B}' '@{0161}' '@{015F}' '@{0165}' '@{017A}'
    '@{017E}' '@{017C}' '@{00DF}' '@{0155}' '@{00E1}' '@{00E2}' '@{0103}' '@{00E4}'
    '@{013A}' '@{0107}' '@{00E7}' '@{010D}' '@{00E9}' '@{0119}' '@{00EB}' '@{011B}'
    '@{00ED}' '@{00EE}' '@{010F}' '@{0111}' '@{0144}' '@{0148}' '@{00F3}' '@{00F4}'
    '@{0151}' '@{00F6}' '@{0159}' '@{016F}' '@{00FA}' '@{0171}' '@{00FC}' '@{00FD}'
    '@{0163}';
```

```
Zcharacter table       ! C3 - Latin 3 (South European)

    '@{0126}' '@{0124}' '@{0130}' '@{015E}' '@{011E}' '@{0134}' '@{017B}' '@{0127}'
    '@{0125}' '@{0131}' '@{015F}' '@{011F}' '@{0135}' '@{017C}' '@{00C0}' '@{00C1}'
    '@{00C2}' '@{00C4}' '@{010A}' '@{0108}' '@{00C7}' '@{00C8}' '@{00C9}' '@{00CA}'
    '@{00CB}' '@{00CC}' '@{00CD}' '@{00CE}' '@{00CF}' '@{00D1}' '@{00D2}' '@{00D3}'
    '@{00D4}' '@{0120}' '@{00D6}' '@{011C}' '@{00D9}' '@{00DA}' '@{00DB}' '@{00DC}'
    '@{016C}' '@{015C}' '@{00DF}' '@{00E0}' '@{00E1}' '@{00E2}' '@{00E4}' '@{010B}'
    '@{0109}' '@{00E7}' '@{00E8}' '@{00E9}' '@{00EA}' '@{00EB}' '@{00EC}' '@{00ED}'
    '@{00EE}' '@{00EF}' '@{00F1}' '@{00F2}' '@{00F3}' '@{00F4}' '@{0121}' '@{00F6}'
    '@{011D}' '@{00F9}' '@{00FA}' '@{00FB}' '@{00FC}' '@{016D}' '@{015D}';

Zcharacter table       ! C4 - Latin 4 (North European)

    '@{0104}' '@{0138}' '@{0156}' '@{0128}' '@{013B}' '@{0160}' '@{0112}' '@{0122}'
    '@{0166}' '@{017D}' '@{0105}' '@{0157}' '@{0129}' '@{013C}' '@{0161}' '@{0113}'
    '@{0123}' '@{0167}' '@{014A}' '@{017E}' '@{014B}' '@{0100}' '@{00C1}' '@{00C2}'
    '@{00C3}' '@{00C4}' '@{00C5}' '@{00C6}' '@{012E}' '@{010C}' '@{00C9}' '@{0118}'
    '@{00CB}' '@{0116}' '@{00CD}' '@{00CE}' '@{012A}' '@{0110}' '@{0145}' '@{014C}'
    '@{0136}' '@{00D4}' '@{00D5}' '@{00D6}' '@{00D8}' '@{0172}' '@{00DA}' '@{00DB}'
    '@{00DC}' '@{0168}' '@{016A}' '@{00DF}' '@{0101}' '@{00E1}' '@{00E2}' '@{00E3}'
    '@{00E4}' '@{00E5}' '@{00E6}' '@{012F}' '@{010D}' '@{00E9}' '@{0119}' '@{00EB}'
    '@{0117}' '@{00ED}' '@{00EE}' '@{012B}' '@{0111}' '@{0146}' '@{014D}' '@{0137}'
    '@{00F4}' '@{00F5}' '@{00F6}' '@{00F8}' '@{0173}' '@{00FA}' '@{00FB}' '@{00FC}'
    '@{0169}' '@{016B}';

Zcharacter table       ! C5 – Cyrillic

    '@{0401}' '@{0402}' '@{0403}' '@{0404}' '@{0405}' '@{0406}' '@{0407}' '@{0408}'
    '@{0409}' '@{040A}' '@{040B}' '@{040C}' '@{040E}' '@{040F}' '@{0410}' '@{0411}'
    '@{0412}' '@{0413}' '@{0414}' '@{0415}' '@{0416}' '@{0417}' '@{0418}' '@{0419}'
    '@{041A}' '@{041B}' '@{041C}' '@{041D}' '@{041E}' '@{041F}' '@{0420}' '@{0421}'
    '@{0422}' '@{0423}' '@{0424}' '@{0425}' '@{0426}' '@{0427}' '@{0428}' '@{0429}'
    '@{042A}' '@{042B}' '@{042C}' '@{042D}' '@{042E}' '@{042F}' '@{0430}' '@{0431}'
    '@{0432}' '@{0433}' '@{0434}' '@{0435}' '@{0436}' '@{0437}' '@{0438}' '@{0439}'
    '@{043A}' '@{043B}' '@{043C}' '@{043D}' '@{043E}' '@{043F}' '@{0440}' '@{0441}'
    '@{0442}' '@{0443}' '@{0444}' '@{0445}' '@{0446}' '@{0447}' '@{0448}' '@{0449}'
    '@{044A}' '@{044B}' '@{044C}' '@{044D}' '@{044E}' '@{044F}' '@{0451}' '@{0452}'
    '@{0453}' '@{0454}' '@{0455}' '@{0456}' '@{0457}' '@{0458}' '@{0459}' '@{045A}'
    '@{045B}' '@{045C}' '@{045E}' '@{045F}';

 Zcharacter table      ! C6 – Arabic

    '@{060C}' '@{061B}' '@{061F}' '@{0621}' '@{0622}' '@{0623}' '@{0624}' '@{0625}'
    '@{0626}' '@{0627}' '@{0628}' '@{0629}' '@{062A}' '@{062B}' '@{062C}' '@{062D}'
    '@{062E}' '@{062F}' '@{0630}' '@{0631}' '@{0632}' '@{0633}' '@{0634}' '@{0635}'
    '@{0636}' '@{0637}' '@{0638}' '@{0639}' '@{063A}' '@{0640}' '@{0641}' '@{0642}'
    '@{0643}' '@{0644}' '@{0645}' '@{0646}' '@{0647}' '@{0648}' '@{0649}' '@{064A}'
    '@{064B}' '@{064C}' '@{064D}' '@{064E}' '@{064F}' '@{0650}' '@{0651}' '@{0652}';

 Zcharacter table       ! C7 – Greek

    '@{0384}' '@{0385}' '@{0386}' '@{0388}' '@{0389}' '@{038A}' '@{038C}' '@{038E}'
    '@{038F}' '@{0390}' '@{0391}' '@{0392}' '@{0393}' '@{0394}' '@{0395}' '@{0396}'
    '@{0397}' '@{0398}' '@{0399}' '@{039A}' '@{039B}' '@{039C}' '@{039D}' '@{039E}'
    '@{039F}' '@{03A0}' '@{03A1}' '@{03A3}' '@{03A4}' '@{03A5}' '@{03A6}' '@{03A7}'
    '@{03A8}' '@{03A9}' '@{03AA}' '@{03AB}' '@{03AC}' '@{03AD}' '@{03AE}' '@{03AF}'
    '@{03B0}' '@{03B1}' '@{03B2}' '@{03B3}' '@{03B4}' '@{03B5}' '@{03B6}' '@{03B7}'
    '@{03B8}' '@{03B9}' '@{03BA}' '@{03BB}' '@{03BC}' '@{03BD}' '@{03BE}' '@{03BF}'
    '@{03C0}' '@{03C1}' '@{03C2}' '@{03C3}' '@{03C4}' '@{03C5}' '@{03C6}' '@{03C7}'
    '@{03C8}' '@{03C9}' '@{03CA}' '@{03CB}' '@{03CC}' '@{03CD}' '@{03CE}';

Zcharacter table       ! C8 – Hebrew

    '@{05D0}' '@{05D1}' '@{05D2}' '@{05D3}' '@{05D4}' '@{05D5}' '@{05D6}' '@{05D7}'
    '@{05D8}' '@{05D9}' '@{05DA}' '@{05DB}' '@{05DC}' '@{05DD}' '@{05DE}' '@{05DF}'
    '@{05E0}' '@{05E1}' '@{05E2}' '@{05E3}' '@{05E4}' '@{05E5}' '@{05E6}' '@{05E7}'
    '@{05E8}' '@{05E9}' '@{05EA}';
```

```
Zcharacter table        ! C9 - Latin 5 (Turkish)
    '@{00C0}' '@{00C1}' '@{00C2}' '@{00C3}' '@{00C4}' '@{00C5}' '@{00C6}' '@{00C7}'
    '@{00C8}' '@{00C9}' '@{00CA}' '@{00CB}' '@{00CC}' '@{00CD}' '@{00CE}' '@{00CF}'
    '@{011E}' '@{00D1}' '@{00D2}' '@{00D3}' '@{00D4}' '@{00D5}' '@{00D6}' '@{00D8}'
    '@{00D9}' '@{00DA}' '@{00DB}' '@{00DC}' '@{0130}' '@{015E}' '@{00DF}' '@{00E0}'
    '@{00E1}' '@{00E2}' '@{00E3}' '@{00E4}' '@{00E5}' '@{00E6}' '@{00E7}' '@{00E8}'
    '@{00E9}' '@{00EA}' '@{00EB}' '@{00EC}' '@{00ED}' '@{00EE}' '@{00EF}' '@{011F}'
    '@{00F1}' '@{00F2}' '@{00F3}' '@{00F4}' '@{00F5}' '@{00F6}' '@{00F8}' '@{00F9}'
    '@{00FA}' '@{00FB}' '@{00FC}' '@{0131}' '@{015F}' '@{00FF}';
```

Reading your source file

As well as populating the ZSCII table, the **C*n*** switches also perform another function; they define the character set which applies to your source file. For example, if you specify **C8**, the compiler expects to find a mixture of English text (Inform words like **Constant**, **if**, **description** and **Initialise**) and Hebrew text (the strings which can be displayed and the dictionary words which can be typed while the game is being played), encoded using ISO 8859-8. Remember we said that values $20..$7F (English letters, digits and punctuation) are identical in each of the nine encodings; the international differences lie in the 96 characters with values $A0..$FF.

More information about ISO 8859 and Microsoft's code pages can be found here

This can cause problems if operating systems don't stick to the ISO 8859 encodings. For example, while Window's Latin 1 code page CP1252 is effectively identical to ISO 8859-1 (which is why games can be written in English, French, German and so on without great difficulty), this doesn't always hold true for other code pages. Thus, the Windows Central European CP1250 is significantly different from ISO 8859-2, and the high values in Windows Cyrillic CP1251 don't match ISO 8859-5 at any point; there may also be discrepancies with other code pages.

If you encounter the problem, you can get round it by creating a mapping file which transforms the character set used by your source file into the ISO 8859 character set which the compiler is expecting. Here are two we prepared earlier: **win1250.map** for use with **C2**:

```
! Windows Central Europe (code page 1250) to ISO 8859-2
C2
  0, 63, 63, 63, 63, 63, 63, 63, 63, 32, 10, 63, 10, 10, 63, 63
 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63
 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63
 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95
 96, 97, 98, 99,100,101,102,103,104,105,106,107,108,109,110,111
112,113,114,115,116,117,118,119,120,121,122,123,124,125,126, 63
 63,129, 44,131, 34, 46, 63, 63,136, 63,169, 60,166,171,174,172
144, 39, 39, 34, 34, 46, 45, 45,152, 84,185, 62,182,187,190,188
 32,183,162,163,164,161,124,167,168, 67,170, 60, 63, 45, 82,175
176, 63,178,179,180, 63, 63, 46,184,177,186, 62,165,189,181,191
192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207
208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223
224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239
240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255
```

and **win1251.map** for use with **C5**:

```
! Windows Cyrillic (code page 1251) to ISO 8859-5
C5
  0, 63, 63, 63, 63, 63, 63, 63, 63, 32, 10, 63, 10, 10, 63, 63
 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63
 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63
 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95
 96, 97, 98, 99,100,101,102,103,104,105,106,107,108,109,110,111
112,113,114,115,116,117,118,119,120,121,122,123,124,125,126, 63
```

```
162,163, 44,243, 34, 46, 63, 63, 63, 63,169, 60,170,172,171,175
242, 39, 39, 34, 34, 46, 45, 45,152, 84,249, 62,250,252,251,255
 32,174,254,168, 36, 63,124,253,161, 67,164, 60, 63, 45, 82,167
 63, 63,166,246, 63, 63, 63, 46,241,240,244, 62,248,165,245,247
176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191
192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207
208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223
224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239
```

Lines starting with "!" are treated as comments. The line beginning with "C" defines the ISO set to map to, and means that you don't then need to provide a **-C*n*** command line switch. To use the mapping, Inform treats each character in the source file as a number between 0 and 255, and uses that number as an index into the mapping table. For example, suppose that the character read in from a Russian Windows source file is the Cyrillic small letter "yu", represented in CP1251 by the number 254. Inform takes that entry in the mapping, which is 238 (highlighted in the win1251.map table above); therefore the "yu" character is regarded as being 238 in ISO 8859-5. Contrast this with a Polish source file, where the character in CP1250 represented by the number 254 -- a small "t" with a cedilla -- is also at position 254 in ISO 8859-2, so the character number remains unchanged.

The name of the mapping file is specified by a compiler path variable **+charset_map**. For example, a Russian game could be compiled with a command line of the form:

```
inform +charset_map=win1251.map +language_name=Russian mygame.inf
```

Printing ZSCII characters

Let's suppose that you've compiled your game without supplying any **C** switch, and that you've used the **Zcharacter** directive to add the three characters '**¢°±**'; these are now in the ZSCII table at positions 224...226, as illustrated in the previous section. You can output them in six ways:

1.	**print "¢°±";**	print a string which includes the literal character values.	The advantage of these forms is readability. The disadvantage is its lack of source portability; the game compiled on the PC will run differently from the same game compiled on the Mac, because the internal character sets of those two machine are not the same. (However, the game compiled on the PC will run the same on both PC and Mac, and vice versa; Inform game files are always portable.)
2.	**print (char) '¢', (char) '°', (char) '±';**	print individual character constants using the literal values.	
3.	**print "@{00A2}@{00B0} @{00B1}";**	print a string which includes the Unicode character values.	The advantage of these forms is that your source code remains portable -- you can for example write the game on a PC, then copy the source to a Mac. The game will compile identically in both environments.
4.	**print (char) '@{00A2}', (char) '@{00B0}', (char) '@{00B1}';**	print individual character constants using the Unicode values.	
5.	**print "@@224@@225 @@226";**	print a string which includes the ZSCII character values.	The disadvantage of these forms is their dependence on the physical ordering of the ZSCII table. Since your additional characters are appended to those taken from ISO 8859, their position varies depending on which **C** switch is used.
6.	**print (char) 224, (char) 225, (char) 226;**	print individual character constants using the ZSCII values.	

Note that forms 1-4 won't compile if you mention a character -- literally or by Unicode value -- which doesn't exist in the game's ZSCII character set. Forms 5 and 6 will compile regardless of whether or not any character has been allocated to the ZSCII value; if there isn't a character at that position, the interpreter prints a question mark.

Remember that you can use constructs like **@^a** to represent '**â**', and like **@ss** to represent '**ß**'. The good news is that these constructs are independent of the **C** switch setting; if '**ß**' appears somewhere in the ZSCII table, then **@ss** will represent it. The bad news is that these constructs aren't extensible. For example, one of the characters loaded by switch **C3** is '**j**' with a circumflex, but you can't use **@^j** to represent it.

> TECHNICAL NOTE: Due to related bugs in the 6.21 compiler and the 0.2 version of the Z-Machine Standards Document, you may encounter problems with printing the left and right guillemet characters. **@{00AB}**, **@@163** and **@<<** should all output '**«**', while **@{00BB}**, **@@162** and **@>>** should all output '**»**'. Although the compiler now gets this right, the result may vary depending on the age of the interpreter which the player is using.

The bottom line is: using these techniques, an Inform game can output a maximum of 191 different characters: the 95 standard values with ZSCII codes of 32...126, and anywhere up to 96 extra characters with ZSCII codes of 155 upwards.

Printing Unicode characters

There's another technique available which bypasses the ZSCII character set (and its limit of 191 characters) altogether -- printing Unicode directly. Version 1.0 the Z-Machine Standards Document introduced the **check_unicode** (test if interpreter can handle a given Unicode character) and **print_unicode** (output a given Unicode character) opcodes. Create this routine:

```
[ Unicode c exist;
    if (HDR_TERPSTANDARD->0 < 1) { @print_char '?'; return; }
    @check_unicode c -> exist;
    if (exist & $0001) @print_unicode c;
    else               @print_char '?';
];
```

> TECHNICAL NOTE: HDR_TERPSTANDARD -- bytes $0032 and $0033 of the game header -- reflects the version of the standard to which the interpreter conforms; we can't use the new opcodes on a pre-1.0 interpreter. If **check_unicode** returns a flag bit of 1 then it's safe to output the character using **print_unicode**.

We can then use this routine either by calling it directly, or using it as a print rule. Its argument is a simple Unicode character number, best given in hexadecimal. For instance, we could display a small Greek 'pi' symbol with either of:

```
Unicode($03C0);
print (Unicode) $03C0;
```

As we said, this method is independent of ZSCII; you can output any characters irrespective of whether they exist in the game's ZSCII character set.

String packing

Although we've been talking about ZSCII as an eight-bit encoding system, the Z-machine doesn't actually hold each ZSCII character in a separate byte. Rather, it stores characters using five-bit units, in which (slightly simplified):

- lower case letters a-z and spaces each occupy a single unit,
- upper case letters A-Z, digits 0-9, newlines and common punctuation symbols each occupy two units, and
- other characters each occupy four units.

The major advantage of this scheme is that lower case text -- the largest component of many games -- is stored fairly economically at three characters to a sixteen-bit word (which is 50% more effective than storing at two bytes to a word). The downside is that each upper case and punctuation character costs ten bits rather than eight, but such characters are relatively infrequent. The major disadvantage -- which doesn't really impinge if you're writing a game in English -- is that each accented character occupies four units; this hits hardest in the dictionary, where only nine storage units are available for each dictionary entry. Entries are stored in lower case, so an entry can comprise up to: nine unaccented letters, one accented letter plus five unaccented letters, or two accented letters plus a single unaccented letter. In a heavily accented language like French, this is a real issue.

There's a way round the limitation, using two more forms of the **Zcharacter** directive in order to manipulate the 'Alphabet table' -- the Z-machine's list of 26 characters which occupy a single storage unit and 51 characters which occupy two units. Here's the default table:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
¤^0123456789.,!?_#'"/\-:()
```

The character shown here as ¤ is a non-printing escape code, while ^ represents the newline character. Using a directive like:

```
  Zcharacter '@`e';
```

near the start of your source file places the specified character '**è**' into the Alphabet table. It does this not by extending the table, but rather by replacing an existing character there which hasn't yet been used; the search for an unused character starts at '**0**' and moves rightwards along the bottom row. You can provide several such **Zcharacter** directives, each one swopping a single character into the table. Alternatively, you can define a completely new Alphabet table by using the fourth and final form of the **Zcharacter** directive; for example:

```
  Zcharacter
      "abcdefghijklmnopqrstuvwxyz"
      "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
      "0123456789-',.;:@'e@`a@`e@`u@^e@^i@^u";
```

In this format, the first and second strings are each exactly 26 characters long, while the third contains only 23 characters -- ¤^" for the escape, the newline and the double quotes are included automatically. The advantage of this method would seem to be that you can choose to retain the digits and the common punctuation, losing instead much rarer characters like '#' and '\'. The resulting table is:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
¤^"0123456789-',.;:éàèùêîû
```

On a sample of French text, the overhead of using accented characters was reduced from 6-7% (using the standard table) to 2-3% (using this table), against the same text without accents.

TECHNICAL NOTE: A default Alphabet table is built into the interpreter. However, if you use either of these **Zcharacter** directives, then the default table becomes inadequate, and the compiler must include a replacement table within the game file. The table's address is given in the game's header at address HDR_ALPHABET-->0. You can display the table by including Roger Firth's `dump.h` library extension, and typing DUMP ALPHA.

Remember that, prior to version 6.30 of the Inform compiler, neither of these **Zcharacter** forms worked properly on most platforms.

> TECHNICAL NOTE: An alternative approach, rather than adding accents to dictionary words, is to remove them from the player's input. For example, you might include code like this in your LanguageToInformese routine:

```
for (i=0 : i<buffer->1 : i++)

     switch (buffer->(i+2)) {
       '@`a','@'a','@^a','@:a','@~a','@oa': buffer->(i+2) = 'a';
       '@ae':     buffer->(i+2) = 'a'; i++; LTI_Insert(i+2,'e');
       '@cc':                                buffer->(i+2) = 'c';
       '@`e','@'e','@^e','@:e':              buffer->(i+2) = 'e';
       '@`i','@'i','@^i','@:i':              buffer->(i+2) = 'i';
       '@~n':                                buffer->(i+2) = 'n';
       '@`o','@'o','@^o','@:o','@~o','@/o': buffer->(i+2) = 'o';
       '@oe':     buffer->(i+2) = 'o'; i++; LTI_Insert(i+2,'e');
       '@ss':     buffer->(i+2) = 's'; i++; LTI_Insert(i+2,'s');
       '@`u','@'u','@^u','@:u':              buffer->(i+2) = 'u';
       '@'y':                                buffer->(i+2) = 'y';
     }
 @tokenise buffer parse;
```

Runtime issues

Access to the full range of 50,000 Unicode characters opens up all sorts of enticing possibilities... but not for long. It's important to remember that the majority of English computer fonts have fairly limited support for characters beyond the ISO 8859-1 repertoire. Sure, if you're based in Algeria, or Bulgaria, or Cambodia, then you'll have appropriate local fonts on your machine. Elsewhere in the world, though, those fonts are unlikely to be available, and therefore a game written to use those fonts will become unplayable. Until such as time as fonts with reasonably complete Unicode support become the norm rather than the exception, you'll either have to be modest in your creativity, or your players will have to be prepared to obtain the fonts for themselves.

A small number of Unicode fonts are readily available. You might consider (see also Alan Wood's site, mentioned earlier, for other possibilities):

- Gentium (free)
- Lucida Sans Unicode (free)
- Code2000 ($5 shareware)

It's not just the lack of appropriate fonts which can be a stumbling block; many Z-machine interpreters themselves are weak in Unicode support. The 1.0 standard requires only that the interpreter be able to display characters up to $00FF -- effectively those defined by the **C1** switch -- which means that, even if the font provides them, you can't rely on characters above that value being displayed. Thus, for example, most ports of the Frotz interpreter assume the use of ISO 8859-1 at all times; they also display '**?**' for *all* Unicode characters in the range $0100-$FFFF, thus precluding the **C2**-**C9** character sets, let alone anything more esoteric. Of the 70 or so Z-machine interpreters in the archive, the only ones offering full support for extended Unicode characters are believed to be Windows Frotz 2002, Zip2000 (RISC OS) and Zoom (X-Windows and Mac OS X); this is alas another argument against being too adventurous in your character displays.

Little-used features: what's a low string?

More information in the DM:§1.11

Whereas most character strings (other than words in the dictionary) are stored all together, along with the Z-code routines, in the High Memory area of the Z-machine, a handful can be held in a part of the Dynamic Memory area called the **Low String Pool**. Although you can't modify those strings as a game progresses, you *can* change the way in which they're accessed. This is best explained by example; suppose that your game includes this statement, which is executed periodically:

```
if (the_time > 359 && the_time < 1080) {
        string 0 "sun"; string 1 "bright blue";
        }    ! day:    06:00 - 17:59
else {
        string 0 "moon"; string 1 "night-time";
        }    ! night: 18:00 - 05:59
```

What happens is: at **compile time** the compiler places all four of those strings "sun", "bright blue", "moon" and "night-time" in the Low String Pool area, along with a default string of "   " (three spaces). At **run time**, the interpreter has 32 Low String pointers, numbered 00 to 31, which initially all point to the default three spaces. When the interpreter executes the statement above, it *either* adjusts pointer 00 to refer to "sun" and pointer 01 to refer to "bright blue", *or* it adjusts pointer 00 to refer to "moon" and pointer 01 to refer to "night-time". Although there's a fixed limit of 32 pointers, there's less restriction on the number of strings to which they point; you can if required adjust the string which each references many times during the course of a game.

What use is this? Well, you can access the current values of the 32 low string pointers by embedding the forms @00 to @31 in print strings; the effect is to output whatever those pointers currently reference. Thus, these statements:

```
style bold; print "The @00 shines down from the @01 sky."; style roman;
```

will produce this 'in daytime':

```
The sun shines down from the bright blue sky.
```

this 'at night':

```
The moon shines down from the night-time sky.
```

and this if you try to print the message before having executed any assignments to pointers 00 and 01 (so be careful):

```
The     shines down from the     sky.
```

In fact, low strings can be useful even if their values *don't* change during the game. Suppose that your family-set game makes frequent mention of "Grandfather". You could code it thus:

```
[ Initialise;
    string 0 "Grandfather";
    ...
];

Object  study "@00's study"
  with  description
          "@00 loves to relax in this shadowy book-filled den, and can be
           found here most evenings, dozing before the big log fire. In all
           the years you've been visiting him, you've never known @00 not to
           have an open book next to his chair.",
        s_to workshop,
  has   light;
```

```
Object  workshop "@00's workshop"
  with  description
            "His workshop is @00's pride and joy, with every tool hanging
             well-oiled in its appointed place, every nut and bolt, nail and
             screw, rivet and washer meticulously filed away in row upon row
             of neatly labelled tobacco tins. @00 can usually be found here
             soon after breakfast, stopping work only at lunchtime and for
             an occasional nap.",
        n_to study,
  has   light;
```

The real advantage here is economy of memory. A word like "Grandfather" occupies 12 storage units (two units for the capital "G" and one for each lower case letter, with each unit requiring five bits) every time it's used. A pointer like "@00" needs only two storage units for each use, plus a single block of 12 units to hold the word in the Low String Pool, so for text that's repeated frequently, the savings can soon add up.

For completeness, we should mention that earlier releases of Inform used a slightly clumsier mechanism, requiring a **Lowstring** directive:

```
Lowstring GF_STR "Grandfather";

[ Initialise;
    string 0 GF_STR;
    ...
];
```

Although this mechanism still works, there's no reason to use it; the new method which doesn't require a **Lowstring** directive is simpler and clearer.

Little-used features: what's an abbreviation?

In the previous topic, we showed how one use for the 32 Low Strings is as shorthand for words or phrases which occur frequently in printed text: you just assign values to them in **Initialise()**, and then use the forms @01..@31 in your text strings, just as you might use **@@64** to print a literal "@", or **@^a** to print "â".

More information in the DM: §45

That's fine for long words or discrete phrases which you'll easily recognise as you're composing your text. However, there are much greater memory savings to be made by abbreviating short sequences, maybe only three or four characters long, which crop up frequently both on their own or as parts of longer words (for example, "the" has already appeared eight times in this topic, "for" six times, " as" four times). Cutting each of those down to two storage units doesn't look much, but *over the length of a big game*, it soon saves a considerable amount. Note that phrase "over the length of a big game" -- all of this talk about conserving memory is relevant only when you're fighting to keep your game within the overall Z-Machine limits (256Kb for a Version 5 game, or more probably 512Kb for a Version 8 game).

Abbreviations are **like** Low Strings in that they're stored in the Low String Pool, referenced within strings using just two storage units, and applicable only to printed text. They're **unlike** Low Strings in four ways:

1. Each abbreviation you wish to use must be pre-declared by an **Abbreviate** directive.
2. You can declare up to 64 abbreviations, versus 32 low strings.
3. Having declared them like that, you don't have to mark them explicitly in the text; instead, the compiler finds them automatically...
4. ...but only if you select Economy mode by supplying the **-e** compiler switch; if you forget that, your **Abbreviate** directives have no effect.

Each potential abbreviation needs to be specified in a separate **Abbreviate** directive, which looks like this:

```
  Abbreviate " the ";
```

This tells the compiler that you think those five characters occur often enough to be worth abbreviating (bit of a no-brainer, really). To decide which phrases are worth abbreviating, you can (a) just guess, (b) run the compiler (very very slowly) with the **-u** switch, or (c) use Emily Short's handy list.

Little-used features: what's a fake action?

As we explained in How do I define a new verb?, an 'action' results from the parser matching some text typed by the player with a line of grammar, and is *by default* dealt with by a matching routine. (We showed a grammar for the verbs SMILE, GRIN, SMIRK, BEAM and TWINKLE causing an action of **Smile**, handled by a **SmileSub()** routine.)

More information in the DM: §6

We emphasise "by default" because it's common to intercept an action before the action routine gets a chance to do its stuff. That's exactly what the **before** property is there for -- to enable an object which is the subject of an action to behave in an appropriate manner, which may be to do something completely non-standard.

Most actions affect a single target object, a few have no target (for example PRAY, SLEEP, SWIM), and a few mention two objects (for example ATTACH X TO Y, PUT X IN Y, THROW X AT Y). It's this last group that we're interested in here; the ones where two objects are involved in the action. Not unreasonably, the second object might like to have a say in what's going on, but there isn't a mechanism equivalent to **before** and **after** -- which apply only to the first object -- whereby the second can automatically be consulted. The closest you get is **react_before** and **react_after**, but any **react_before** properties run *prior to* any **before** properties, when what you'd really like is something that runs once the first object's **before** has decided to go ahead, but before it actually happens.

So, if there isn't a built-in way of dealing with situations like this, you have to devise something appropriate yourself. By way of example, let's consider the POINT X AT Y process which we described in How do I use my new object property?, where the first object uses this code to get the second object involved:

```
  before [;
    PointAt:
      if (self has light && second provides when_lit)
          return second.when_lit();
  ],
```

and the second object deals with that involvement something like this:

```
  when_lit "The owl blinks in mild surprise.",
```

Here's that same example, with an alternative implementation using a **fake action**. What's 'fake' about such actions is that they never appear in a line of grammar (and so cannot be directly triggered by anything the player types); neither do they have a handler routine. Since the presence of both those is what makes a normal action acceptable, you need to persuade the compiler that what you're doing is valid; that's what the **Fake_Action** directive is there for.

The whole thing looks like this:

```
  Fake_Action PointedAt;
```

```
Object  -> "flashlight"
  with  name 'flashlight',
        before [;
          PointAt:
            if (self has light) {
                action = ##PointedAt;
                if (second.before()) { action = ##PointAt; rtrue; }
                action = ##PointAt;
            }
        ],
        after [;
          SwitchOn:  give self light;
          SwitchOff: give self ~light;
        ],
  has   switchable ~on ~light;

Object  -> "owl"
  with  name 'owl',
        before [;
          PointedAt:
            "The owl blinks in mild surprise.";
        ],
  has   animate;

Object  -> "mouse"
  with  name 'mouse',
        before [;
          PointedAt:
            remove self;
            "The startled mouse disappears into the undergrowth.";
        ],
  has   animate;

[ PointAtSub; "Pointless."; ];

Verb 'aim' 'point'
    * held 'at'/'on'/'towards' noun -> PointAt;

Extend only 'shine'
    * held 'at'/'on'/'towards' noun -> PointAt;
```

The verb stuff at the end is the same as before, but the highlighted object definitions are a bit different. The flashlight's **before** property now has to adjust the **action** variable, effectively pretending that the current action is PointedAt rather than PointAt, then manually give the second object's **before** property a chance to react, then finally reinstate the real PointAt action before letting the action complete appropriately. That's all a bit messy; on the other hand, the objects being pointed at can now deal with the situation by means of regular **before** properties rather than with our invented **when_lit** property. (By the way, if you're wondering why we didn't use the <PointedAt second> or <<PointedAt second>> statements in the flashlight, it's because neither of those returns the second object's response, which the flashlight needs to determine whether (if true) the action has been dealt with, or (if false) the default action should be allowed to continue.)

Personally, I think that fake actions have no advantage over the method using local properties. Also, the DM4 labels them "obsolete", so that's a better reason for avoid their use.

How is the parse array structured?

Every time that the player types a command, the parser:

1. stores all of the typed characters in a **buffer** array (and, in the Z-machine, converts them to lower case),

2. divides that stream of characters into 'tokens' -- commas, periods, and words separated by spaces,
3. attempts to find each token in the game's dictionary, and
4. stores the information about the tokens in a second array: **parse**.

For example, the command SAILOR,SAY HELLO is divided into four tokens: SAILOR, the comma, SAY and HELLO.

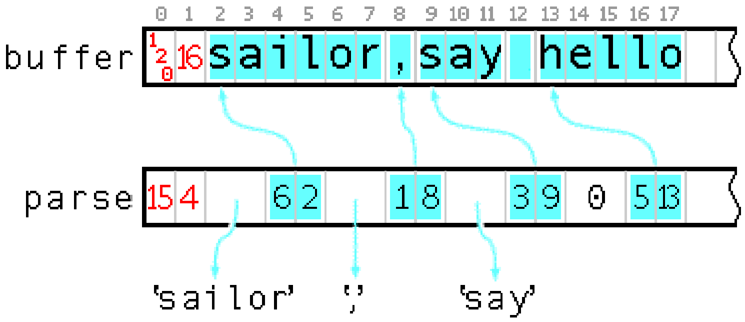
The library routine **KeyboardPrimitive()** is responsible for this processing, and you can call it yourself if you need to ask the player for some additional input. Even if you don't, there are still cases where it's useful to be able to manipulate the input command stream (see, for example, the use of the **BeforeParsing()** entry point in this topic), which requires that you understand the structure of the two arrays.

The DM4 is a little confused on the arrays' layout, so we'll explain them using our SAILOR,SAY HELLO example. After that command, the contents of the two arrays are as shown below. **buffer** is a byte array holding the characters as typed. The first byte is preset to the maximum number of characters that can be handled, and shouldn't be altered -- the library sets it to 120. The second byte specifies that 16 characters have been typed. Those character then follow.

More information in the DM: §2.5

So, that gives us:

`buffer->0` is always 120
`buffer->1` is the character count (16)
`buffer->2` is the first character ('s')
`buffer->3` is the second character ('a')
...
`buffer->17` is the sixteenth character ('o').



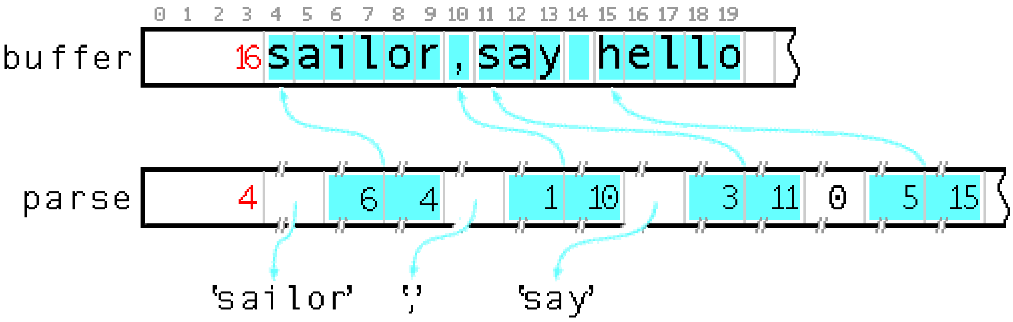
parse is more complex, being a mixed array of bytes and words. The first byte is preset to the maximum number of tokens that can be handled, and shouldn't be altered -- the library sets it to 15. The second byte specifies that 4 tokens have been identified. Information on those tokens then follows.

In this example:

`parse->0` is always 15
`parse->1` is the token count (4)
`parse-->1` is the dictionary address of the first token ('sailor')
`parse->4` is the number of characters in the first token (6)
`parse->5` is the token's start position in **buffer** (2)
`parse-->3` is the dictionary address of the second token (',')
`parse->8` is the number of characters in the second token (1)
`parse->9` is the token's start position in **buffer** (8)
`parse-->5` is the dictionary address of the third token ('say')
`parse->12` is the number of characters in the third token (3)
`parse->13` is the token's start position in **buffer** (9)
`parse-->7` is zero, because the fourth token ('hello') isn't in the dictionary

`parse->16` is the number of characters in the fourth token (5)
`parse->17` is the token's start position in **buffer** (13).

If you're using Glulx, the principles are the same... and the arrays are different (see the Glulx Technical Reference for more details). Neither **buffer** nor **parse** stores its maximum capacity at the beginning. Also, the Glulx **buffer** uses an initial word rather than a byte to hold the number of character typed; after that, its content is identical to the Z-machine's. The Glulx version of **parse**, however, differs from the Z-machine's in that it's a word-only array -- one word for the token count, and then one each for dictionary address, token length and token start position in **buffer**. (to save space on this page, the width of entries in the diagram on the right has been compressed.) Note that **buffer** contains the characters exactly as typed, without any case conversion.



To step through all of the parsed tokens, use a loop like this:

```
[ tokenCount;
    #Ifdef TARGET_ZCODE;
    return parse->1;
    #Ifnot; ! TARGET_GLULX
    return parse-->0;
    #Endif; ! TARGET_
];

for (w=1 : w<=tokenCount() : w++) { ... }
```

where variable `w` numbers the tokens as 1,2,3... Then, use these routines to fetch the three attributes of each token:

```
[ tokenDict w;  ! dictionary value of token 1,2,3...
    #Ifdef TARGET_ZCODE;
    return parse-->(2*w - 1);
    #Ifnot; ! TARGET_GLULX
    return parse-->(3*w - 2);
    #Endif; ! TARGET_
];

[ tokenLen w;   ! length in chars of token 1,2,3...
    #Ifdef TARGET_ZCODE;
    return parse->(4*w);
    #Ifnot; ! TARGET_GLULX
    return parse-->(3*w - 1);
    #Endif; ! TARGET_
];

[ tokenPos w;   ! position in buffer of token 1,2,3...
    #Ifdef TARGET_ZCODE;
    return parse->(4*w + 1);
    #Ifnot; ! TARGET_GLULX
    return parse-->(3*w);
    #Endif; ! TARGET_
];
```

Here's a simple example: an intransigent NPC who responds to all commands with "Surely you don't expect me to...?". This is his **orders** property:

```
  orders [ bufsize i;
      #Ifdef TARGET_ZCODE;
      bufsize = WORDSIZE + buffer->1;              ! end of Z-machine buffer
      #Ifnot; ! TARGET_GLULX
      bufsize = WORDSIZE + buffer-->0;             ! end of Glulx buffer
      #Endif; ! TARGET_
      if (tokenDict(2) == ',//') i = tokenPos(3); ! NPC, do something
      else                       i = tokenPos(4); ! TELL NPC TO do something
      print "~Surely you don't expect me to ";
      for (: i<bufsize : i++) print (char) buffer->i;
      "?~";
  ],
```

The only real complexity is the need to distinguish between NPC, *DO SOMETHING* (where the actual command starts at the third token) and TELL NPC TO *DO SOMETHING* (where the fourth token begins the command). For example:

```
  >SAILOR,JUMP IN THE AIR
  "Surely you don't expect me to jump in the air?"

  >TELL SAILOR TO LIE ON THE GROUND
  "Surely you don't expect me to lie on the ground?"

  >YES
  That was a rhetorical question.
```

How are the standard print rules implemented?

A print rule is simply a routine which outputs its single argument in some form. For example, a print rule for small Roman numerals might look like this:

```
  [ roman n;
      switch (n) {
        1:  print "I";
        2:  print "II";
        3:  print "III";
        4:  print "IV";
        5:  print "V";
        6:  print "VI";
        7:  print "VII";
        8:  print "VIII";
        9:  print "IX";
        10: print "X";
        default: print "[", n, "]";
      }
  ];
```

Having defined such a routine, you can either use it as a print rule or call it directly -- the two work identically:

```
  print  "Chapter  ",  (roman)  chapNumber,  ":  ",  (string)  chapTitle,  "^";  !
Option 1

  print "Chapter ";
  roman(chapNumber);                                                              !
Option 2
  print ": ", (string) chapTitle, "^";
```

Caution: remember that, as we explained much earlier, you'll get a spurious '1' appearing if you call the print rule as a routine:

```
  print "Chapter ", roman(chapNumber), ": ", (string) chapTitle, "^";  ! WRONG
```

Although it's normally shorter and more convenient to invoke your routine as a rule in a **print** statement (Option 1) rather than as a conventional call (Option 2), there are exceptional cases. One such is when you wish to print to an array rather than to the screen, for example so that you can manipulate the characters prior to display. The library routine **PrintToBuffer()** does just this, so one way of outputting the chapter number in lower-case roman numerals is this:

```
Constant MYBUF_SIZE 10;
Array myBuf buffer MYBUF_SIZE;

n = PrintToBuffer(myBuf, MYBUF_SIZE, roman, chapNumber);
for (i=0 : i<n : i++) print (char) LowerCase(myBuf->(i+WORDSIZE));
```

All well and good, but what if you wish to invoke one of the standard print rules -- **(a)**, **(name)**, **(The)** and so on -- in this manner? There's a problem, in that although each rule is implemented by a routine, the name of the routine isn't the same as the name of the rule. So, to help if you wish to use the standard rules in this manner, here are the routines to call (the "__" are *two* underscore characters):

Standard print rule	Equivalent routine call
(A) *object*	**CIndefArt(***object***)**
(a) *object*	**IndefArt(***object***)**
(address) *dictionary_word*	**RT__ChPrintA(***dictionary_word***)**
(char) *expression*	**RT__ChPrintC(***expression***)**
(name) *object*	**PrintShortName(***object***)**
(number) *expression*	**EnglishNumber(***expression***)**
(object) *object*	**RT__ChPrintO(***object***)**
(property) *property*	**Print__PName(***property***)**
(string) *string*	**RT__ChPrintS(***string***)**
(The) *object*	**CDefArt(***object***)**
(the) *object*	**DefArt(***object***)**

11 · Tips and Techniques (advanced)

(For issues which are even more esoteric than these, visit the Inform Patch List.)

What's the associativity of ~/~~?

There's a problem in the way that Inform handles the two NOT operators in an expression, in that they don't work quite as you might anticipate:

More information in the DM: §1.6 §1.8

<table>
<tr><th>If you write these:</th><th>...they're taken as:</th><th>...rather than as:</th></tr>
<tr><td><code>~X&amp;Y or ~X|Y</code></td><td><code>~(X&amp;Y) or ~(X|Y)</code></td><td><code>(~X)&amp;Y or (~X)|Y</code></td></tr>
<tr><td><code>~~X&amp;&amp;Y or ~~X||Y</code></td><td><code>~~(X&amp;&amp;Y) or ~~(X||Y)</code></td><td><code>(~~X)&amp;&Y or (~~X)||Y</code></td></tr>
</table>

You need to include the parentheses (as shown in the rightmost column) to be sure of getting the 'correct' result.

How do I write an XOR function?

Inform doesn't come with built-in XOR (Exclusive OR) operators, but you can achieve the effect with these routines:

```
[ XOR a b; return (a | b) & (~(a & b)); ];         ! Bitwise Exclusive OR
[ XXOR a b; return (a || b) && (~~(a && b)); ];    ! Logical Exclusive OR
```

Can I use expressions in <...>?

The **<...>** statement (to trigger an action) and the **<<...>>** statement (to trigger an action and return **true**) are usually invoked with a literal 'action' argument followed by zero, one or two variable names, as in:

```
<<Inv>>;
<<Examine self>>;
<<Insert noun second>>;
```

More information in the DM: §6

These work fine; what *isn't* so successful is trying to use a variable for the first argument, or expressions for the second and third arguments. The trick is simple: just enclose things in parentheses **(...)**. For example:

```
<<(action) noun (parent(self))>>;
```

Is there a Library routine to print direction names?

Yes: **LanguageDirection()**, which take a direction property (like **e_to**) as its argument:

More information in the DM: §37

```
LanguageDirection(property);
```

If you're starting from a direction object (like **e_obj**), just change the call to this:

```
LanguageDirection(object.door_dir);
```

Can I introduce a short time delay?

More information in the DM: §42

You may occasionally want your game to pause for a short period, for example between characters appearing tickertape-style (rather than as the normal instantaneous output stream). The Z-machine is not well-equipped to handle time delays, but *on some interpreters* you can achieve the effect using the **@read_char** assembler statement; here's one approach:

```
[ Sleep x; if (x) @read_char 1 x Sleep ->x; ];
```

The routine's argument *x* is the length of the required delay in tenths of a second; for example, Sleep(20) will pause for about two seconds. A side-effect of using **@read_char** is that the player can press any key to immediately terminate the delay -- useful for those with experience and/or a low boredom threshold. This example prints the characters in a string array, with an optional delay after each:

More information in the DM: §2.4

```
Array welcome_msg string "Welcome to the game!";

[ PrintStringArray the_array the_delay i;
        for (i=1 : i<=the_array->0 : i++) {
            print (char) the_array->i;
            Sleep(the_delay);
            };
        ];

[ Initialise;
        location = farmyard;
        new_line; PrintStringArray(welcome_msg, 5); "^";
        ];
```

If you're using Glulx you have access to a more powerful and flexible system of timed events, though it's slightly more trouble to set up. In addition to a Sleep() routine, you need also to define an event handler, something like this:

```
[ Sleep x;      ! For compatibility, delay in 10ths of a second
        if (glk_gestalt(gestalt_Timer) && x > 0) {
            glk_request_timer_events(x*100);
            KeyCharPrimitive();
            glk_request_timer_events(0);
            }
        ];

[ HandleGlkEvent ev context abortres;
      switch (ev-->0) {
          evtype_Timer:
              if (context == 1) glk_cancel_char_event(gg_mainwin);
              else              glk_cancel_line_event(gg_mainwin, GLK_NULL);
              abortres-->0 = 0;
              return 2;
          }
      ];
```

Why can't I address "Dr.Jekyll" or "Mr.Hyde"?

The Inform parser treats "." and "THEN" as separators between commands, so these three examples all do the same thing:

```
> X DAGGER
> TAKE IT

>X DAGGER THEN TAKE IT
>X DAGGER.TAKE IT
```

More information in the DM: §2.5 §30

This handling of "." becomes a problem if you wish to use it as part of a dictionary word, for example in "DR." or "MR." to indicate an abbreviation. The easiest way round the difficulty is to supply a **BeforeParsing()** entry point, which is called after the parser has read in some text and set up the **buffer** (raw text) and **parse** (parsed text) arrays, but before doing anything else. Here's a suggested routine:

```
[ BeforeParsing x;
    for (wn=2 : wn<parse->1 : wn++)
        if (buffer->(parse->(4*wn + 1)) == '.' &&
          parse-->(2*wn - 3) == 'dr' or 'mr' or 'mrs' or 'prof' or
          'rev' or 'st' && parse-->(2*wn + 1) == 'jekyll' or 'hyde') {
              buffer->(parse->(4*wn + 1)) = ' ';
              x++;
        }
    if (x) @tokenise buffer parse;
];
```

In this example:

- **wn** is used to index the **parse** array, omitting the first and last entries
- **parse->(4*wn)** is the length in characters of entry **wn** in the **parse** array
- **parse->(4*wn + 1)** is the index of the start of that entry in the **buffer** array
- **parse-->(2*wn - 3)** is the dictionary value of entry **wn-1**
- **parse-->(2*wn + 1)** is the dictionary value of entry **wn+1**

The routine scans the **parse** array. If it finds three consecutive entries *X*.*Y*, where *X* is "DR" or "MR" and *Y* is "JEKYLL" or "HYDE" then it overwrites the "." in the raw text **buffer** with a space, and calls **@tokenise** to re-create the **parse** array from the modified **buffer**. The NPC's **name** property handles the outcome in the usual way:

```
Object  "Dr Jekyll"
   with  name 'dr' 'jekyll' 'doctor',
         ...
   has   animate proper;
```

How do I right-align printed numbers?

When printing, Inform treats numbers as signed decimal values, and outputs the digits left-aligned (for example "123"). If you want to insert leading spaces or zeroes to force right-alignment (giving " 123" or "00123"), for example when printing a column of figures, you could use this routine:

```
[ AlignNum
    num     ! number to be aligned (unsigned 0-65535)
    pad     ! right-align by inserting this char (default is left-align)
    wid     ! field width for right-alignment (default is 5)
    d e;    ! local variables
    d = 5;
    if (wid == 0) wid = 5;
    if (num >= 0)
        switch (num) {
             0 to    9: d = 1;
            10 to   99: d = 2;
           100 to  999: d = 3;
          1000 to 9999: d = 4;
        }
    else {
        num = (num & $7FFF)*2 - (num + 30000);
        for (e=3 : UnsignedCompare(num,9999)==1 : num=num-10000) e++;
        switch (num) {
             0 to    9: e = e * 1000;
            10 to   99: e = e * 100;
           100 to  999: e = e * 10;
        }
    }
    if (pad) for (d = wid-d : d>0 : d--) print (char) pad;
    if (e) print e;
    print num;
];
```

The first parameter is the number to be printed. Note that this is treated as an *unsigned* value, in the range 0-65535. The second parameter is a single character. It's optional, typically ' ' or '0', and is inserted as initial padding to cause the number to be right-aligned. If this parameter is omitted, the number is left-aligned. The third parameter is a number. It's optional, and specifies the width of the right-aligned number. If this parameter is omitted, the default width is five characters. If the value that you specify is too small, the number overflows; it is *not* truncated to fit the specified width.

To display a four-digit safe combination, where any of the digits could be zero, you might produce a line like "The safe combination is '0107'." by these three statements:

```
print "The safe combination is '";
AlignNum(safe.combination,'0',4);
print "'.^";
```

You can't use **AlignNum()** directly as a print rule, because it usually requires three parameters, and a print rule handles only one. Instead, you could create a separate print rule as a routine which calls **AlignNum()**, and use it within a single print statement like this:

```
[ AlignZ4 n; AlignNum(n,'0',4); ];

print "The safe combination is '", (AlignZ4) safe.combination, "'.^";
```

Can an 'achieved task' have a negative score?

Inform offers two built-in scoring systems: a simpler one of awarding points for collecting objects and visiting rooms, and a more advanced scheme which keeps track of which 'tasks' the player has accomplished. In this second system, the value of each scored task must lie in the range 0-255. Occasionally, you'd like to be able to award a score higher than that, or to penalize the player by *deducting* points -- that is, awarding a negative score for a prohibited task. For this to work, the **task_scores** array, which is where you define the score for each defined task 0...*N*-1, must be configured as a word array rather than the current byte array. That is, instead of creating the array like this:

More information in the DM: §22

```
Constant TASKS_PROVIDED;
Constant NUMBER_TASKS 5;
Constant MAX_SCORE 25;

Array task_scores -> 3 7 3 5 7;
```

you create an array of words:

```
Array task_scores --> 3 7 3 5 7;
```

However, there's more to it than that; you also need to fix those parts of the Library which are currently expecting a byte array. You do this within your game file by Replacing two Library routines:

```
Replace Achieved;
Replace FullScoreSub;
...
Include "VerbLib";
...
[ Achieved num;
  if (task_done->num==0)
  {   task_done->num=1;
      score = score + task_scores-->num;     ! CHANGED
  }
];

[ FullScoreSub i;
  ScoreSub();
  if (score==0 || TASKS_PROVIDED==1) rfalse;
  new_line;
  L__M(##FullScore,1);

  for (i=0:i<NUMBER_TASKS:i++)
      if (task_done->i==1)
      {   PANum(task_scores-->i);            ! CHANGED
          PrintTaskName(i);
      }

  if (things_score~=0)
  {   PANum(things_score); L__M(##FullScore,2); }
  if (places_score~=0)
  {   PANum(places_score); L__M(##FullScore,3); }
  new_line; PANum(score); L__M(##FullScore,4);
];
```

Those two routines are copied from `verblibm.h`, and in each case a single line needs to be changed (so that **task_scores->** becomes **task_scores-->**).

Fortunately, version 6/11 of the Library makes this much simpler. Those accesses to the byte array are isolated in a little routine, which you simply Replace:

```
Replace TaskScore;

[ TaskScore i; return task_scores-->i; ];
```

Once you've done that, by either method, you can happily award large and negative scores (the latter enclosed in parentheses):

```
Array task_scores --> 300 700 300 (-100) 700;
```

Are the points awarded by a 'scored' object adjustable?

More information in the DM: §22

In the built-in scoring system of awarding points for collecting objects and visiting rooms, the score is determined by the values of the constants **OBJECT_SCORE** (default of 4) for the first time you pick up an object with a **scored** attribute, and **ROOM_SCORE** (default of 5) for entering a **scored** room.

If you wish for greater flexibility, it's very straightforward to enhance the two Library routines responsible for this system, by creating a **scored_value** individual property:

```
Replace NoteObjectAcquisitions;
Replace ScoreArrival;
...
Include "Parser";
Include "VerbLib";
...
[ NoteObjectAcquisitions i s;
    objectloop (i in player)
        if (i hasnt moved) {
            give i moved;
            if (i has scored) {
                if (i provides scored_value) s = i.scored_value();
                else                         s = OBJECT_SCORE;
                score = score + s;
                things_score = things_score + s;
            }
        }
];

[ ScoreArrival s;
    if (location hasnt visited) {
        give location visited;
        if (location has scored) {
            if (location provides scored_value) s = location.scored_value();
            else                                s = ROOM_SCORE;
            score = score + s;
            places_score = places_score + s;
        }
    }
];
```

Those two routines are copied from parserm.h and verblibm.h respectively, and in each case the value of a new local variable 's' is set either to the object or room's **scored_value** property (if it has such a property) or to the appropriate constant (which is what currently happens). **scored_value** can be a numeric value -- the number of points to award -- or a routine which returns such a number. Here are a couple of examples:

```
Object  puzzle_room "Puzzle chamber"
  with  description "A bare room.",
        scored_value 10,
  has   scored light;

Object  -> box "box"
  with  name 'box',
        with_key key,
  has   scored container openable ~open lockable locked;

Object  -> key "key"
  with  name 'key',
        scored_value [;
            if (box has moved) return 2;
            print "^[You'd have scored more by taking the box first.]^";
            return 1;
        ],
  has   scored;
```

Note that the 'box' object has a **scored** attribute but no **scored_value** property, and so will award the default number of points on first being TAKEn (a **scored_value** property with no associated **scored** attribute is ignored). And the result is...

```
Puzzle chamber
A bare room.

You can see a box (which is closed) and a key here.

[Your score has just gone up by ten points.]

>GET KEY
Taken.

[You'd have scored more by taking the box first.]

[Your score has just gone up by one point.]

>GET BOX
Taken.

[Your score has just gone up by four points.]
```

ChooseObjects() is messing up TAKE ALL. What can I do?

Inform provides a **ChooseObjects()** entry point so that you can influence how the parser chooses between ambiguous object names, typically because you've some additional knowledge about the objects and their situation to which the parser isn't privy. In general, it's better to keep any **ChooseObjects()** routine short and specific, instead controlling object selection using an object's **parse_name** property wherever possible.

ChooseObjects() is called in two situations: once (with the *code* argument equal to 2) to influence which objects are preferred matches to a noun phrase by returning a rank from 0 to 9; and then at a later stage, if the player has typed ALL, it is called with code 1 or 0 to choose whether to accept or reject individual objects. A generic routine would look like this:

More information in the DM: §33

```
[ ChooseObject obj code;
    switch (code) {
     2: ! Parser wants an 'appropriateness' rating forobj
        ! Inspect obj, and then...
        return 0;   ! Sorry -- can't offer any guidance, or
        return 1;   ! Very slight preference forobj, or
        ...
        return 9;   ! Very strong preference forobj.
     1, ! Parser proposes to include obj in ALL.
     0: ! Parser proposes to exclude obj from ALL.
        ! Inspect obj, and then...
        return 0;   ! Agree with Parser, or
        return 1;   ! Force inclusion, or
        return 2:   ! Force exclusion.
    }
];
```

If both of these methods are employed at the same time, this can lead to **ChooseObjects(obj,1)** being called only for objects to which **ChooseObjects(obj,2)** has just given a high ranking. If the individual object(s) proposed in that way are then rejected (by returning 2), then ALL doesn't refer to anything and a "There are none at all available!" message results.

The way to separate these two processes is to insert this line in your **ChooseObjects()**:

```
[ ChooseObject obj code;
    switch (code) {
     2: ! Parser wants an 'appropriateness' hint forobj
        if (indef_wanted == 100) return 0;
        ! Inspect obj, and then...
        ...
];
```

This checks an undocumented global variable to see if the parser is handling an ALL and will shortly be proposing inclusions/exclusions. The resulting refusal to give an 'appropriateness' rating means that TAKE ALL will no longer be affected.

Can ChooseObjects() tell if it's evaluating noun or second?

As explained in the previous topic, the **ChooseObjects()** entry point can be used to influence which objects are preferred matches to a noun phrase. Here, we'll show how the first object chosen (the variable **noun**) can affect the second object chosen (the variable **second**). We'll illustrate this usage by inventing a couple of new verbs:

```
[ LoosenSub;  "You can't loosen that!"; ];
[ TightenSub; "You can't tighten that!"; ];

Verb 'loosen'  * noun 'with' held  -> Loosen;
Verb 'tighten' * noun 'with' held  -> Tighten;
```

The idea is that the first object -- we'll be using a 10mm brass bolt -- can be loosened or tightened by use of the second object, an implement of some form. Here are three possible tools (though only two of them are capable of doing the job):

```
Object  spanner10 "10mm spanner"
  with  name '10mm' 'spanner' 'tool';

Object  spanner15 "15mm spanner"
  with  name '15mm' 'spanner' 'tool';

Object  wrench "adjustable wrench"
  with  name 'adjustable' 'wrench' 'tool';
```

And here is the skeleton of our brass bolt, together with a useful print rule routine:

```
Object  "10mm brass bolt"
  with  name '10mm' 'brass' 'bolt',
        description [;
             print_ret (The) self, " is currently ", (showState) self, ".";
        ],
        tightness ##Tighten,
        with_tools spanner10 wrench,
        ...
  has   static;

[ showState o;
      if (o.tightness == ##Tighten) print "tight";
      else print "loose"; ];
```

You'll notice that we've invented two individual properties. **tightness** indicates the current state of the bolt, defined as one of the action values ##Tighten (the bolt is currently tight) or ##Loosen (it's currently loose). More interesting is **with_tools**, which contains a list of the tools which can be used to adjust the bolt: the 10mm spanner and the adjustable wrench are both acceptable, but the 15mm spanner isn't listed, and so won't do the job.

Here's the full definition, with the addition of the **before** property which does the actual work of changing the bolt's state:

```
Object  "10mm brass bolt"
  with  name '10mm' 'brass' 'bolt',
        description [;
             print_ret (The) self, " is currently ", (showState) self, ".";
        ],
          tightness ##Tighten,
          with_tools spanner10 wrench,
          before [ i n;
            Loosen, Tighten:
              if (self.tightness == action)
                  print_ret (The) self, " is already ", (showState) self, ".";
              n = self.#with_tools / WORDSIZE; ! number of possible tools
              for (i=0 : i<n : i++)
                  if (self.&with_tools-->i == second or -second) {
                      self.&with_tools-->i = -second;
                      self.tightness = action;
                      print_ret (The) self, " is now ", (showState) self, ".";
                  }
              print_ret (The) second, " doesn't seem to fit ", (the) self, ".";
          ],
    has   static;
```

This isn't as complex as it looks. The property is preventing the bolt from being tightened unless it's currently loose and vice versa, and then checking that the **second** object is one of the acceptable tools listed in the **with_tools** property (if it's not, the "doesn't seem to fit" message appears). If an acceptable tool is employed, the bolt's state is changed (by `self.tightness = action;`). The only slightly odd thing is the statement `self.&with_tools-->i = -second;`, which updates the list of acceptable tools to mark the one that the player has actually used. Why have we done that? You'll see in a minute or two.

All of the code that we've written so far will work just as it is. However, since the point of this topic is to illustrate the benefits that **ChooseObjects()** can offer, we'd better bring such a routine into play. This is what we want it to do, when evaluating potential tool objects for **second**: (1) prefer objects in the **noun.with_tools** list to ones which aren't listed, (2) prefer objects at the start of the list to those at the end, and (3) prefer objects which have already been used to those which haven't. Here's the code to do all that:

```
  [ ChooseObjects obj code
      i n o;
      if (code ~= 2) rfalse; ! Agree with Parser's ALL decisions
      ! Give an 'appropriateness' score to obj
      if (parameters > 0 && action_to_be == ##Loosen or ##Tighten) {
          ! Is obj a suitable tool?
          o = inputobjs-->2; ! object to be Loosened/Tightened
          if (o ofclass Object && o provides with_tools) {
              n = o.#with_tools / WORDSIZE; ! number of possible tools
              for (i=0 : i<n : i++) ! tool used already, in order of list
                  if (o.&with_tools-->i == -obj) return max(9-i, 7); !9, 8, 7, 7...
              for (i=0 : i<n : i++) ! tool not yet used, in order of list
                  if (o.&with_tools-->i == obj) return max(6-i, 1); !6, 5, 4, 3, 2,
                                                                     !1, 1...
          }
      }
      return 0; ! not Loosen/Tighten, or not a listed tool
  ];

  [ max a b; if (a < b) return b; return a; ];
```

When evaluating candidates for **second** in the context of Loosen/Tighten, this routine returns a ranking of 6 for the 10mm spanner (first in the list), and 5 for the wrench (next in the list); if more tools were listed, they'd get 4, then 3, then 2, then any further tools would all rank as 1. The 15mm spanner isn't in the with_tools list, so gets the default rank of 0. However, once the 10mm spanner has been used its ranking increases to 9, while the wrench would rank as 8 after succcessful use. The effect of all this is that the player can now type simply LOOSEN BOLT with a strong probability that the parser can infer which tool is intended, without having to ask "Which did you mean...?".

We've highlighted two undocumented variables in the definition of ChooseObjects(), and they're the key to making this work. **parameters** holds 0 while the routine is evaluating candidates for **noun**, and 1 while evaluating **second**; in the latter case the array element **inputobjs-->2** holds the object previously chosen as **noun**. However, be warned: this all works *only* for simple grammars such as our Loosen and Tighten verbs. It *can't* be used in grammars which employ the **multiexcept** or **multiinside** tokens, or the **reverse** keyword, because in these cases **noun** is evaluated *after* **second**, and so isn't available to influence the decision-making process.

How can I change the size of a string or table array?

When you declare a **string** (or **table**) array, Inform stores the array's element count in the first byte (or word). For example, this declaration:

More information in the DM: §2.4

```
Array ourPets table "cat" "dog" "rabbit" "guineapig";
```

creates an array with five elements. ourPets-->1 hold the address of the string "cat"; ourPets-->2 refers to "dog", ourPets-->3 to "rabbit" and ourPets-->4 to "guineapig". In addition, the compiler places the count of the number of entries -- 4 in this example -- in ourPets-->0.

The problem comes should you try to modify that 'count' value at runtime. For example, suppose the kids grow up and change their taste in pets. You might write this code:

```
ourPets-->3 = "pony";
ourPets-->2 = "conger eel";
ourPets-->1 = "tarantula";
ourPets-->0 = 3;
```

Unfortunately, it isn't that easy. On the Z-machine in Strict mode, you'll get a runtime error from the ourPets-->0 = 3; statement; Glulx gives a *compiler* error for the same thing. Inform is being a little over-protective here; there's no reason why you shouldn't be able to change the count value, providing the new value is something sensible, but alas you can't.

Well, you can; you've just got to cheat a little. The easiest way round the problem is to conceal the fact that you're updating a **string** or **table** array's count by writing a couple of tiny routines:

```
[ StoreStringSize arr val; arr->0 = val; ];

[ StoreTableSize arr val; arr-->0 = val; ];
```

then you can replace ourPets-->0 = 3; with StoreTableSize(ourPets,3); to perform the assignment without upsetting the compiler.

The alternative technique is to drop into assembly language -- slightly more efficient, slightly more trouble. The code you need is one of these:

```
@storeb  arr 0 val; ! arr->0  = val (Z-machine)
@storew  arr 0 val; ! arr-->0 = val (Z-machine)
@astoreb arr 0 val; ! arr->0  = val (Glulx)
@astore  arr 0 val; ! arr-->0 = val (Glulx)
```

Can I prompt the player to key in some information?

More information in the DM: §2.5 §42

As the DM4 says, Inform's support for reading from the keyboard is fairly limited. If you wish to bypass the parser and handle keyboard input directly, you've only two choices, and both are imperfect: you can use the **read** statement (or the **@aread** assembly language statement) to input a line of text, or the **@read_char** assembly language statement (or the **KeyCharPrimitive()** routine, which also works with Glulx) to accept a single character:

- **read** puts a line of characters into a buffer, displaying each character as it is typed, and handling all keystokes until the player terminates the input by pressing the Return key. Good news: the Backspace key can be used to correct typing mistakes. Bad news: all alphabetic characters are converted to lower case.
- **@read_char** returns a single keystroke, without displaying anything. Good news: no case conversion takes place. Bad news: you can echo the character back to the player, but you can't Backspace to erase a character once you've echoed it.

If you're asking the player to key in her name, location or other relatively lengthy string, then on balance it's probably more important to offer the editing convenience of Backspace than it is to maintain the case of what she types. So, here's a routine, and its Glulx equivalent, to read a line of characters from the keyboard. Note that the characters are placed in a **buffer** array -- one whose first *word* contains the number of characters that have been typed, and whose subsequent *bytes* contain those characters. The routine returns the number of typed characters:

```
#Ifdef TARGET_ZCODE;

[ KeyLine buf max;
    buf->0 = max;
    buf->1 = 0;
    read buf 0;
    buf->0 = 0;
    return buf-->0;
];

#Ifnot; ! TARGET_GLULX

[ KeyLine buf max;
    glk($00D0, gg_mainwin, buf+WORDSIZE, max, 0); ! request_line_event
    while (true) {
        glk($00C0, gg_event); ! select
        if (gg_event-->0 == 3 && gg_event-->1 == gg_mainwin) break;
    }
    buf-->0 = gg_event-->2;
    return buf-->0;
];

#Endif; ! TARGET_
```

Here's how you use the routine to ask the player to type in her name:

```
Constant playerName_SIZE 20;
Array    playerName buffer playerName_SIZE;

do
    print "What's your name? ";
until (KeyLine(playerName, playerName_SIZE));
```

The **do** loop repeats the prompt until the player types something. And here's a routine to display the name:

```
[ PrintString buf
    i;
    for (i=0 : i<buf-->0 : i++)
        print (char) buf->(i+WORDSIZE);
];
```

You can use this either as a printing routine, or as a print rule:

```
print "Hello, ";
PrintString(playerName);
".";

"Hello, ", (PrintString) playerName, ".";
```

It's time to think about case conversion; the current effect (on the Z-machine) isn't quite right:

```
What's your name? Mary Jane
Hello, mary jane.
```

This routine capitalises the first letter of each word in the string -- suitable if the string is a name (Paul O'Brian, Jean-Paul Satre) or a location (Weston-Super-Mare, Los Angeles):

```
[ CapitaliseString buf
    i c flg;
    for (i=0,flg=true : i<buf-->0 : i++) {
        c = buf->(i+WORDSIZE);
        if (c >= 'a' && c <= 'z') {
            if (flg) buf->(i+WORDSIZE) = UpperCase(c);
            flg = false;
        }
        else
            flg = true;
    }
];
```

Here's how to test whether two **buffer** strings are identical:

```
[ CompareStrings bufA bufB
    i;
    if (bufA-->0 ~= bufB-->0) rfalse;
    for (i=0 : i<bufA-->0 : i++)
        if (Lowercase(bufA->(i+WORDSIZE)) ~= LowerCase(bufB->(i+WORDSIZE)))
            rfalse;
    rtrue;
];
```

And finally, all of the routines working together:

```
Constant playerName_SIZE 20;
Array    playerName buffer playerName_SIZE;
Array    extraName1 buffer "Roger";
Array    extraName2 buffer "Sonja";

do
    print "What's your name? ";
until (KeyLine(playerName, playerName_SIZE));
CapitaliseString(playerName);
if (CompareStrings(playerName, extraName1) ||
    CompareStrings(playerName, extraName2))
    "Welcome, oh honoured one.";
else
    "Hello, ", (PrintString) playerName, ".";
```

Note, in passing, that our "What's your name" prompt ends with a "? "; you could also use ">> ", but it's best to prevent confusion with the parser's prompt by *not* using just "> ".

How do I put single characters into the dictionary?

The most common way of adding to the dictionary is via an object's **name** property. A typical **name** property looks like this:

```
name 'xray' 'x-ray' 'camera' 'machine',
```

but if you want to add just one character, you have to be a little devious. For example, to add a dash on its own you can't simply include **'-'** in the list, because single characters in single quotes are always treated as ZSCII character constants (even in **name** properties, where that makes no logical sense). So instead, use any of these:

```
name '-//' ...
name '@{2D}//'...
name "-" ...
```

in which 2D is the Unicode hexadecimal value for a dash. The first of those three forms is generally preferred, *except* where the character you're trying to add is a forward slash. In that case, use either of the other two forms:

```
name '@{2F}//' ...
name "/" ...
```

Why doesn't 'my' work in a name property?

The parser treats MY (also HIS, HER, ITS and THEIR) as special descriptors, in roughly the same way as THE, A and SOME are handled automatically. This works nicely most of the time; consider a simple piece of code:

```
Object  -> "red hat"
  with  name 'red' 'hat',
  has   clothing;

Object  -> "blue hat"
  with  name 'blue' 'hat',
  has   clothing;
```

Here, you can see how MY HAT is taken to mean the hat in the player's possession (and by implication, HAT without MY means any hats *not* being held):

```
You can see a red hat and a blue hat here.

>GET HAT
Which do you mean, the red hat or the blue hat?

>RED
Taken.

>EXAMINE HAT
(the blue hat)
You see nothing special about the blue hat.

>EXAMINE MY HAT
You see nothing special about the red hat.
```

Sometimes, though, you'd like MY to reflect ownership rather than current possession. Perhaps the blue hat is a family heirloom or a badge of office: you want to refer to it as MY HAT *whether or not you're holding it*. There's slightly more to this than just adding 'my' to the blue hat's **name** property; you also need to allow for the parser's automatic handling (which consumes the MY before considering the words listed in **name**). A tiny **parse_name** property does the trick:

```
Object  -> "blue hat"
  with  name 'my' 'blue' 'hat',
        parse_name [;
            if (parser_action == ##TheSame) return -2;
            if (indef_type & MY_BIT) wn--;
            return -1;
        ],
  has   clothing;
```

And now you get this:

```
>GET RED HAT
Taken.

>EXAMINE HAT
(the blue hat)
You see nothing special about the blue hat.

>EXAMINE MY HAT
You see nothing special about the blue hat.

>EXAMINE RED HAT
You see nothing special about the red hat.
```

Note that this still leaves MY referring to objects in your possession if there's no match against a **name** property. If you think that'll be confusing, you can disable it by adding to your **Initialise()**:

```
LanguageDescriptors-->1 = 'this';
```

How can the player input numbers bigger than 10000?

One of the standard grammar tokens is **number**, which matches any decimal value in the range 0..10000. Sometimes you may need to use values larger than this, or less than zero, so here's a general parsing routine **AnyNumber()**, based on Exercise 92 in the DM4, which works with the full range of decimal numbers -- and their binary and hexadecimal equivalents -- for both Z-machine and Glulx.

More information in the DM: §31

```
#Ifdef TARGET_ZCODE;                ! decimal range is -32768 to 32767
Constant MAX_DECIMAL_SIZE 5;
Constant MAX_DECIMAL_BASE 3276;
#Ifnot; ! TARGET_GLULX          ! decimal range is -2147483648 to 2147483647
Constant MAX_DECIMAL_SIZE 10;
Constant MAX_DECIMAL_BASE 214748364;
#Endif; ! TARGET_

[ AnyNumber wa we
    sign base digit digit_count num;

    if (wa == 0) { wa = WordAddress(wn); we = WordLength(wn); }
    we = wa + we;
    sign = 1; base = 10;
    if     (wa->0 == '-') { sign = -1; wa++; }
    else {
        if (wa->0 == '$') { base = 16; wa++; }
        if (wa->0 == '$') { base = 2;  wa++; }
    }
    if (wa >= we) return GPR_FAIL;  ! no digits after -/$
    while (wa->0 == '0') wa++;      ! skip leading zeros
    for (num=0,digit_count=1 : wa<we : wa++,digit_count++) {
        switch (wa->0) {
          '0' to '9': digit = wa->0 - '0';
          'A' to 'F': digit = wa->0 - 'A' + 10;
          'a' to 'f': digit = wa->0 - 'a' + 10;
          default:    return GPR_FAIL;
        }
        if (digit >= base) return GPR_FAIL;
        switch (base) {
          16:     if (digit_count > 2*WORDSIZE)  return GPR_FAIL;
          2:      if (digit_count > 8*WORDSIZE)  return GPR_FAIL;
          10:
            if (digit_count >  MAX_DECIMAL_SIZE) return GPR_FAIL;
            if (digit_count == MAX_DECIMAL_SIZE) {
                if (num >  MAX_DECIMAL_BASE)     return GPR_FAIL;
                if (num == MAX_DECIMAL_BASE) {
                    if (sign == 1  && digit > 7) return GPR_FAIL;
                    if (sign == -1 && digit > 8) return GPR_FAIL;
                }
            }
        }
        num = base*num + digit;
    }
    parsed_number = num * sign;
    wn++;
    return GPR_NUMBER;
];
```

In accordance with the rules governing general parsing routines, **AnyNumber()** returns either the constant GPR_NUMBER if it matches a valid number (whose value is in the library variable **parsed_number**), or the constant GPR_FAIL if there is no match. You might use **anynumber** in a replacement for the standard SET grammar (note that **special** is an undocumented token, described in 1996 by the DM3 as "obsolete and best avoided", which matches either a decimal value or a dictionary word)

```
Verb 'set' 'adjust'
    * noun                      -> Set
    * noun 'to' special         -> SetTo;
```

with a version accepting the full range of numbers:

```
Extend 'set' replace
    * noun                      -> Set
    * noun 'to' anynumber       -> SetTo;
```

You could implement addition and subtraction in a similar manner:

```
[ IncreaseSub; "You can't increase that!"; ];

[ DecreaseSub; "You can't decrease that!"; ];

Verb 'increase' 'increment' 'inc'
    * noun 'by' anynumber        -> Increase;

Verb 'add'
    * anynumber 'to' noun        -> Increase reverse;

Verb 'decrease' 'decrement' 'dec'
    * noun 'by' anynumber        -> Decrease;

Verb 'subtract'
    * anynumber 'from' noun      -> Decrease reverse;
```

The job of turning a string of decimal digits into a single numeric value is handled by the library routine **TryNumber()**, which is where the 0..10000 limit originates. One way of bypassing this limitation is by defining a **ParseNumber()** entry point routine, but this turns out to be slightly less useful than you'd think: if **ParseNumber()** rejects a string like "99999" (because 32767 is the Z-machine's limit) then **TryNumber()** accepts and processes it, returning the truncated value 10000.

An alternative approach is simply to Replace **TryNumber()** with your own code. A definition which leverages the **AnyNumber()** routine shown earlier is:

```
[ TryNumber wordnum
    x y z;

    ! accept "one" to "twenty"
    y = wn; wn = wordnum; z = NextWord(); wn = y;
    z = NumberWord(z); if (z) return z;

    ! if provided, use ParseNumber() entry point
    x = WordAddress(wordnum); y = WordLength(wordnum);
    z = ParseNumber(x, y); if (z) return z;

    ! accept any integer value
    if (AnyNumber(x, y) == GPR_FAIL) return -1000;
    wn--;
    return parsed_number;
];
```

The advantage of this approach is that the (very nearly) full range of parsed numeric values is now available everywhere using the existing **number** token, without you needing to replace verb grammars or make other code changes. The disadvantage is that "-1000" can't be input, since that's the value which **TryNumber()** returns to indicate a non-numeric value, and we have to retain this for compatibility with existing library code (the use of **anynumber** as a grammar token avoids this problem).