Chapter VI: Using the Compiler

> I was promised a horse, but what I got instead
> was a tail, with a horse hung from it almost dead.
>
> — Palladas of Alexandria (319?–400?),
> translated by Tony Harrison (1937–)

§38 Controlling compilation from within

Inform has a number of directives for controlling which pieces of source code are compiled: for instance, you can divide your source code into several files, compiled together or separately, or you can write source code in which different passages will be compiled on different occasions. Most of these directives are seldom seen, but almost every game uses:

```
Include "filename";
```

which instructs Inform to glue the whole of that source code file into the program right here. It is exactly equivalent to removing the `Include` directive and replacing it with the whole file `"filename"`. (The rules for how Inform interprets `"filename"` vary from machine to machine: for instance, it may automatically add an extension such as "`.inf`" if your operating system normally uses filename extensions and it may look in some special directory. Run Inform with the `-h1` switch for further information.) Note that you can write

```
Include ">shortname";
```

to mean "the file called `"shortname"` which is in the same directory that the present file came from". This is convenient if all the files making up the source code of your game are housed together.

Next, there are a number of "conditional compilation" directives. They take the general form of a condition:

`Ifdef` ⟨name⟩`;`	Is ⟨name⟩ defined as having some meaning?
`Ifndef` ⟨name⟩`;`	Is ⟨name⟩ undefined?
`Iftrue` ⟨condition⟩`;`	Is this ⟨condition⟩ true?
`Iffalse` ⟨condition⟩`;`	Is this ⟨condition⟩ false?

followed by a chunk of Inform and then, optionally,

```
Ifnot;
```

and another chunk of Inform; and finally

```
Endif;
```

At this point it is perhaps worth mentioning that (most) directives can also be interspersed with statements in routine declarations, provided they are preceded by a # sign. For example:

```
[ MyRoutine;
#Iftrue MAX_SCORE > 1000;
  print "My, what a long game we're in for!^";
#Ifnot;
  print "Let's have a quick game, then.^";
#Endif;
  PlayTheGame();
];
```

which actually only compiles one of the two `print` statements, according to what the value of the constant `MAX_SCORE` is.

△ One kind of "if-defined" manoeuvre is so useful that it has an abbreviation:

`Default` ⟨name⟩ ⟨value⟩`;`

defines ⟨name⟩ as a constant if it wasn't already the name of something: so it's equivalent to

`Ifndef` ⟨name⟩`; Constant` ⟨name⟩ `=` ⟨value⟩`; Endif;`

Similarly, though far less often used, `Stub <name> <number>;` defines a do-nothing routine with this name and number (0 to 3) of local variables, if it isn't already the name of something: it is equivalent to

`Ifndef` ⟨name⟩`; [` ⟨name⟩ `x1 x2 ... x`⟨number⟩`; ]; Endif;`

. . . . .

Large standard chunks of Inform source code are often bundled up into "libraries" which can be added to any Inform story file using the `Include` directive. Almost all Inform adventure games include three library files called "Parser", "VerbLib" and "Grammar", and several dozen smaller libraries have also been written. Sometimes, though, what you want to do is "include all of this library file except for the definition of `SomeRoutine`". You can do this by declaring:

```
Replace SomeRoutine;
```

before the relevant library file is included. You still have to define your own `SomeRoutine`, hence the term "replace".

△△ How does Inform know to ignore the `SomeRoutine` definition in the library file, but to accept yours as valid? The answer is that a library file is marked out as having routines which can be replaced, by containing the directive

```
System_file;
```

All eight of the standard Inform library files (the three you normally `Include` in games, plus the five others which they `Include` for you) begin with this directive. It also has the effect of suppressing all compilation warnings (but not errors) arising from the file.

. . . . .

One way to follow what is being compiled is to use the `Message` directive. This makes the compiler print messages as it compiles:

```
Message "An informational message";
Message error "An error message";
Message fatalerror "A fatal error message";
Message warning "A warning message";
```

Errors, fatal errors and warnings are treated as if they had arisen from faults in the source code in the normal way. See §40 for more about the kinds of error Inform can produce, but for now, note that an error or fatal error will prevent any story file from being produced, and that messages issued by `Message warning` will be suppressed if they occur in a "system file" (one that you have marked with a `System_file` directive). Informational messages are simply printed:

```
Message "Geometry library by Boris J. Parallelopiped";
```

prints this text, followed by a new-line.

△ One reason to use this might be to ensure that a library file fails gracefully if it needs to use a feature which was only introduced on a later version of the Inform compiler than the one it finds itself running through. For example:

```
Ifndef VN_1610;
Message fatalerror
    "The geometry extension needs Inform 6.1 or later";
Endif;
```

By special rule, the condition "`VN_1610` is defined" is true if and only if the compiler's release number is 6.10 or more; similarly for the previous releases 6.01, first to include message-sending, 6.02, 6.03, 6.04, 6.05, 6.10, which expanded numerous previous limits on grammar, 6.11, 6.12, which allowed Inform to read from non-English character sets, 6.13, 6.15, which allowed parametrised object creation, 6.20, which introduced strict error checking, and finally (so far) 6.21, the first to feature Infix. A full history can be found in the *Technical Manual.*

. . . . .

Inform also has the ability to link together separately-compiled pieces of story file into the current compilation. This feature is provided primarily for users with slowish machines who would sooner not waste time compiling the standard Inform library routines over and over again. Linking isn't something you can do entirely freely, though, and if you have a fast machine you may prefer not to bother with it: the time taken to compile a story file is now often dominated by disc access times, so little or no time will be saved.

The pieces of pre-compiled story file are called "modules" and they cannot be interpreted or used for anything other than linking.

The process is as follows. A game being compiled (called the "external" program) may `Link` one or more pre-compiled sections of code called "modules". Suppose the game Jekyll has a subsection called Hyde. Then these two methods of making Jekyll are, very nearly, equivalent:

(1) Putting `Include "Hyde";` in the source for `"Jekyll"`, and compiling `"Jekyll"`.
(2) Compiling `"Hyde"` with the `-M` ("module") switch set, then putting `Link "Hyde";` into the same point in the source for `"Jekyll"`, and compiling `"Jekyll"`.

Option (2) is faster as long as `"Hyde"` does not change very often, since its ready-compiled module can be left lying around while `"Jekyll"` is being developed.

Because "linking the library" is by far the most common use of the linker, this is made simple. All you have to do is compile your game with the `-U`

switch set, or, equivalently, to begin your source code with

```
Constant USE_MODULES;
```

This assumes that you already have pre-compiled copies of the two library modules: if not, you'll need to make them with

```
inform -M library/parserm.h
inform -M library/verblibm.h
```

where `library/parserm.h` should be replaced with whatever filename you keep the library file "`parserm`" in, and likewise for "`verblibm`". This sounds good, but here are four caveats:

(1) You can only do this for a game compiled as a Version 5 story file. This is the version Inform normally compiles to, but some authors of very large games need to use Version 8. Such authors usually have relatively fast machines and don't need the marginal speed gain from linking rather than including.
(2) It's essential not to make any `Attribute` or `Property` declarations *before* the `Include "Parser"` line in the source code, though *after* that point is fine. Inform will warn you if you get this wrong.
(3) Infix debugging, `-X`, is not compatible with linking, and strict error checking `-S` does not apply within modules.
(4) The precompiled library modules always include the `-D` debugging verbs, so when you come to compile the final release version of a game, you'll have to compile it the slow way, i.e., without linking the library.

△△ If you intend to write your own pre-compilable library modules, or intend to subdivide a large game into many modular parts, you will need to know what the limitations are on linking. (In the last recourse you may want to look at the *Technical Manual*.) Here's a brief list:

(1) The module must make the same `Property` and `Attribute` directives as the main program and in the same order. Including the library file `"linklpa.h"` ("link library properties and attributes") declares the library's own stock, so it might be sensible to do this first, and then include a similar file defining any extra common properties and attributes you need.
(2) The module cannot contain grammar (i.e., use `Verb` or `Extend` directives) or create "fake actions".
(3) The module can only use global variables defined outside the module if they are explicitly declared before use using the `Import` directive. For example, writing `Import global frog;` allows the rest of the module's source code to refer to the

variable `frog` (which must be defined in the outside program). Note that the Include file `"linklv.h"` ("link library variables") imports all the library variables, so it would be sensible to include this.

(4) An object in the module can't inherit from a class defined outside the module. (But an object outside can inherit from a class inside.)

(5) Certain constant values in the module must be known at module-compile-time (and must not, for instance, be a symbol only defined outside the module). For instance: the size of an array must be known now, not later; so must the number of duplicate members of a `Class`; and the quantities being compared in an `Iftrue` or `Iffalse`.

(6) The module can't: define the `Main` routine; use the `Stub` or `Default` directives; or define an object whose parent object is not also in the same module.

These restrictions are mild in practice. As an example, here is a short module to play with:

```
Include "linklpa";        ! Make use of the properties, attributes
Include "linklv";         ! and variables from the Library
[ LitThings x;
  objectloop (x has light)
      print (The) x, " is currently giving off light.^";
];
```

It should be possible to compile this `-M` and then to `Link` it into another game, making the routine `LitThings` exist in that game.

. . . . .

Every story file has a release number and a serial code. Games compiled with the Inform library print these numbers in one line of the "banner". For instance, a game compiled in December 1998 might have the banner line:

Release 1 / Serial number 981218 / Inform v6.20 Library 6/8

The release number is 1 unless otherwise specified with the directive

```
Release <number>;
```

This can be any Inform number, but convention is for the first published copy of a game to be numbered 1, and releases 2, 3, 4, ... to be amended re-releases.

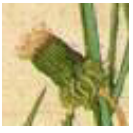
The serial number is set automatically to the date of compilation in the form "yymmdd", so that `981218` means "18th December 1998" and `000101` means "1st January 2000". You can fix the date differently by setting

```
Serial "dddddd";
```

where the text must be a string of 6 (decimal) digits. Inform's standard example games do this, so that the serial number will be the date of last modification of the source code, regardless of when the story file is eventually compiled.

△ The Inform release number is written into the story file by Inform itself, and you can't change it. But you can make the story file print out this number with the special statement `inversion;`.

§39 Controlling compilation from without



The Inform compiler has the equivalent of a dashboard of "command line switches", controlling its behaviour. Most of these have only two settings, off or on, and most of them are off most of the time. Others can be set to a number between 0 and 9. In this book switches are written preceded by a minus sign, just as they would be typed if you were using Inform from the command line of (say) Unix or RISC OS. Setting -x, for instance, causes Inform to print a row of hash signs as it compiles:

```
inform -x shell
RISC OS Inform 6.20 (10th December 1998)
::##############################################################
#################
```

One hash sign is printed for every 100 textual lines of source code compiled, so this row represents about eight thousand lines. (During the above compilation, carried out by an Acorn Risc PC 700, hashes were printed at a rate of about thirty per second.) -x is provided not so much for information as to indicate that a slow compilation is continuing normally. Contrariwise, the -s switch offers more statistics than you could possibly have wanted, as in the following monster compilation (of 'Curses'):

```
RISC OS Inform 6.20 (10th December 1998)
In: 25 source code files           17052 syntactic lines
 22098 textual lines              860147 characters (ISO 8859-1 Latin1)
Allocated:
  1925 symbols (maximum 10000)   1022182 bytes of memory
Out:   Version 8 "Extended" story file 17.981218 (343.5K long):
    37 classes (maximum 64)          579 objects (maximum 639)
   169 global vars (maximum 233)    4856 variable/array space (m. 8000)
   153 verbs (maximum 200)          1363 dictionary entries (m. 2000)
   318 grammar lines (version 2)     490 grammar tokens (unlimited)
   167 actions (maximum 200)          34 attributes (maximum 48)
    51 common props (maximum 62)     153 individual props (unlimited)
267892 characters used in text    195144 bytes compressed (rate 0.728)
     0 abbreviations (maximum 64)    891 routines (unlimited)
 25074 instructions of Z-code      10371 sequence points
 52752 bytes readable memory used (maximum 65536)
351408 bytes used in Z-machine    172880 bytes free in Z-machine
Completed in 8 seconds
```

The complete list of switches is listed when you run Inform with `-h2`, and also in Table 3 at the back of this book, where there are notes on some of the odder-looking ones.

When the statistics listing claims that, for instance, the maximum space for arrays is 10,000, this is so because Inform has only allocated enough memory to keep track of 10,000 entries while compiling. This in turn is because Inform's "memory setting" `$MAX_STATIC_DATA` was set to 10,000 when the compilation took place.

Between them, the "memory settings" control how much of your computer's memory Inform uses while running. Too little and it may not be able to compile games of the size you need, but too much and it may choke any other programs in the computer for space. It's left up to you to adjust the memory settings to suit your own environment, but most people leave them alone until an error message advises a specific increase.

Finally, Inform has "path variables", which contain filenames for the files Inform uses or creates. Usage of these varies widely from one operating system to another. Run Inform with the `-h1` switch for further information.

. . . . .

Inform's switches, path variables and memory settings are set using "Inform Command Language" or ICL. The usual way to issue ICL commands, at least on installations of Inform which don't have windowed front-ends, is to squeeze them onto the command line. The standard command line syntax is

`inform` ⟨ICL commands⟩ ⟨source file⟩ ⟨output file⟩

where only the ⟨source file⟩ is mandatory. By default, the full names to give the source and output files are derived in a way suitable for the machine Inform is running on: on a PC, for instance, `advent` may be understood as asking to compile `advent.inf` to `advent.z5`. This is called "filename translation". No detailed information on filenaming rules is given here, because it varies so much from machine to machine: see the `-h1` on-line documentation. Note that a filename can contain spaces if it is written in double-quotes and if the operating system being used can cope with spaces (as Mac OS can, for instance).

To specify sprawling or peculiar projects may need more ICL commands than can fit comfortably onto the command line. One possible ICL command, however, is to give a filename in (round) brackets: e.g.,

```
inform -x (skyfall_setup) ...
```

which sets the `-x` switch, then runs through the text file `skyfall_setup` executing each line as an ICL command. This file then look like this:

```
! Setup file for "Skyfall"
-d                    ! Contract double spaces
$max_objects=1000     ! 500 of them snowflakes
(usual_setup)         ! include my favourite settings, too
+module_path=mods     ! keep modules in the "mods" directory
```

ICL can include comments if placed after `!`, just as in Inform. Otherwise, an ICL file has only one command per line, with no dividing semicolons. Using ICL allows you to compile whole batches of files as required, altering switches, path variables and memory settings as you go. Notice that ICL files can call up other ICL files, as in the example above: don't allow a file to call up another copy of itself, or the compilation will all end in tears.

△ When typing such a command into a shell under a Unix-like operating systems, you may need to quote the parentheses:

```
inform -x '(skyfall_setup)' ...
```

This instructs the shell to pass on the command literally to Inform, and not to react to unusual characters like `$`, `?`, `(` or `)` in it. The same may be needed for other ICL commands such as:

```
inform -x '$MAX_OBJECTS=2000' ...
```

△ Windowed front-ends for Inform sometimes work by letting the user select various options and then, when the "Go" button is pressed, convert the state of the dialogue box into an ICL file which is passed to Inform.

△△ If you need to use Inform without benefit of either a command line or a fancy front-end, or if you want your source code to specify its own switch settings, you can still set (most) switches by placing the directive

```
Switches ⟨some settings⟩;
```

at the very beginning of your source code. (So that you can still override such settings, the switch `-i` tells Inform to ignore all `Switches` directives.)

. . . . .

The ICL commands are as follows:

-⟨switches⟩
Set these switches; or unset any switch preceded by a tilde ~. (For example, -a~bc sets a, unsets b and sets c.)

$list
List current memory settings.

$?⟨name⟩
Ask for information on what this memory setting is for.

$small
Set the whole collection of memory settings to suitable levels for a small game.

$large
Ditto, for a slightly larger game.

$huge
Ditto, for a reasonably big one.

$⟨name⟩=⟨quantity⟩
Alter the named memory setting to the given level.

+⟨name⟩=⟨filename⟩
Set the named pathname variable to the given filename, which should be one or more filenames of directories, separated by commas.

compile ⟨filename⟩ ⟨filename⟩
Compile the first-named file, containing source code, writing the output program to the (optional) second-named file.

(⟨filename⟩)
Execute this ICL file (files may call each other in this way).

. . . . .

It's a nuisance to have to move all the memory settings up or down to cope with a big game or a small computer, so $small, $large and $huge are provided as short cuts. Typically these might allocate 350K, 500K or 610K respectively. Running

```
inform $list
```

will list the various settings which can be changed, and their current values. Thus one can compare small and large with:

```
inform $small $list
inform $large $list
```

If Inform runs out of allocation for something, it will generally print an error message like:

```
"Game", line 1320: Fatal error: The memory setting MAX_OBJECTS
(which is 200 at present) has been exceeded.  Try running Inform
again with $MAX_OBJECTS=<some-larger-number> on the command line.
```

and it would then be sensible to try

```
inform $MAX_OBJECTS=250 game
```

which tells Inform to try again, reserving more memory for objects this time. ICL commands are followed from left to right, so

```
inform $small $MAX_ACTIONS=200 ...
```

will work, but

```
inform $MAX_ACTIONS=200 $small ...
```

will not because the `$small` changes `MAX_ACTIONS` back again. Changing some settings has hardly any effect on memory usage, whereas others are expensive to increase. To find out about, say, `MAX_VERBS`, run

```
inform $?MAX_VERBS
```

(note the question mark) which will print some very brief comments.

§40 Error messages

Five kinds of error are reported by Inform: a fatal error is a breakdown severe enough to make Inform stop working at once; an error allows Inform to continue for the time being, but will normally cause Inform not to output the story file (because it is suspected of being damaged); and a warning means that Inform suspects you may have made a mistake, but will not take any action itself. The fourth kind is an ICL error, where a mistake has been made in a file of ICL commands for Inform to follow: an error on the command line is called a "command line error" but is just another way to get an ICL error. And the fifth is a compiler error, which appears if Inform's internal cross-checking shows that it is malfunctioning. The text reporting a compiler error asks the user to report it the author of Inform.

Fatal errors

1. *Too many errors*

```
Too many errors: giving up
```

After 100 errors, Inform stops, in case it has been given the wrong source file altogether, such as a program written in a different language.

. . . . .

2. *Input/output problems*

Most commonly, Inform has the wrong filename:

```
Couldn't open source file ⟨filename⟩
Couldn't open output file ⟨filename⟩
```

(and so on). More seriously the whole process of file input/output (or "I/O") may go wrong for some reason to do with the host computer: for instance, if it runs out of disc space. Such errors are rare and look like this:

```
I/O failure: couldn't read from source file
I/O failure: couldn't backtrack on story file for checksum
```

and so forth. The explanation might be that two tasks are vying for control of the same file (e.g., two independent Inform compilations trying to write a debugging information file with the same name), or that the file has somehow been left "open" by earlier, aborted compilation. Normally you can only have

at most 256 files of source code in a single compilation. If this limit is passed, Inform generates the error

```
Program contains too many source files: increase #define
    MAX_SOURCE_FILES
```

This might happen if file A includes file B which includes file C which includes file A which... and so on. Finally, if a non-existent pathname variable is set in ICL, the error

```
No such path setting as ⟨name⟩
```

is generated.

. . . . .

3. *Running out of things*

If there is not enough memory even to get started, the following appear:

```
Run out of memory allocating ⟨n⟩ bytes for ⟨something⟩
Run out of memory allocating array of ⟨n⟩x⟨m⟩ bytes for ⟨something⟩
```

More often memory will run out in the course of compilation, like so:

```
The memory setting ⟨setting⟩ (which is ⟨value⟩ at present) has
    been exceeded. Try running Inform again with $⟨setting⟩=⟨larger-value⟩
    on the command line.
```

(For details of memory settings, see §39 above.) In a really colossal game, it is just conceivable that you might hit

```
One of the memory blocks has exceeded 640K
```

which would need Inform to be recompiled to get around (but I do not expect anyone ever to have this trouble, because other limits would be reached first). Much more likely is the error

```
The story file/module exceeds version ⟨n⟩ limit (⟨m⟩K) by ⟨b⟩ bytes
```

If you're already using version 8, then the story file is full: you might be able to squeeze more game in using the `Abbreviate` directive, but basically you're near to the maximum game size possible. Otherwise, the error suggests that you might want to change the version from 5 to 8, and the game will be able to grow at least twice as large again. It's also possible to run out not of story file space but of byte-accessible memory:

```
This program has overflowed the maximum readable-memory size of the
    Z-machine format. See the memory map below: the start of the
    area marked "above readable memory" must be brought down to
    $10000 or less.
```

Inform then prints out a memory map so that you can see what contributed to the exhaustion: there is detailed advice on this vexatious error in §45. Finally, you can also exhaust the number of classes:

```
Inform's maximum possible number of classes (whatever amount of memory
    is allocated) has been reached. If this causes serious
    inconvenience, please contact the author.
```

At time of writing, this maximum is 256 and nobody has yet contacted the author.

Errors

In the following, anything in double-quotes is a quotation from your source code; other strings are in single-quotes. The most common error by far takes the form:

```
Expected ⟨...⟩ but found ⟨...⟩
```

There are 112 such errors, most of them straightforward to sort out, but a few take some practice. One of the trickiest things to diagnose is a loop statement having been misspelt. For example, the lines

```
pritn "Hello";
While (x == y) print "x is still y^";
```

produce one error each:

```
line 1: Error: Expected assignment or statement but found pritn
line 2: Error: Expected ';' but found print
```

The first is fine. The second is odd: a human immediately sees that `While` is meant to be a `while` loop, but Inform is not able to make textual guesses like this. Instead Inform decides that the code intended was

```
While (x == y); print "x is still y^";
```

with `While` assumed to be the name of a function which hasn't been declared yet. Thus, Inform thinks the mistake is that the `;` has been missed out.

In that example, Inform repaired the situation and was able to carry on as normal in subsequent lines. But it sometimes happens that a whole cascade of errors is thrown up, in code which the user is fairly sure must be nearly right. What has happened is that one syntax mistake threw Inform off the right track, so that it continued not to know where it was for many lines in a row. Look at the first error message, fix that and then try again.

. . . . .

1. *Reading in the source-code*

```
Unrecognised combination in source: ⟨text⟩
Alphabetic character expected after ⟨text⟩
Name exceeds the maximum length of ⟨number⟩ characters: ⟨name⟩
Binary number expected after '$$'
Hexadecimal number expected after '$'
Too much text for one pair of quotations '...' to hold
Too much text for one pair of quotations "..." to hold
No text between quotation marks ''
Expected 'p' after '//' to give number of dictionary word ⟨word⟩
```

In that last error message, "number" is used in the linguistic sense, of singular versus plural.

. . . . .

2. *Characters*

```
Illegal character found in source: ⟨char⟩
No such accented character as ⟨text⟩
'@{' without matching '}'
At most four hexadecimal digits allowed in '@{...}'
'@{...}' may only contain hexadecimal digits
'@..' must have two decimal digits
Character can be printed but not input: ⟨char⟩
Character can be printed but not used as a value: ⟨char⟩
Alphabet string must give exactly 23 [or 26] characters
Character can't be used in alphabets unless entered into Zcharacter
    table: ⟨char⟩
Character must first be entered into Zcharacter table: ⟨char⟩
Character can only be used if declared in advance as part of
    'Zcharacter table': ⟨char⟩
Character duplicated in alphabet: ⟨char⟩
No more room in the Zcharacter table
```

Characters are given by ISO Latin-1 code number if in this set, and otherwise by Unicode number, and are also printed if this is possible. For instance, if you try to use an accented character as part of an identifier name, you cause an error like so:

```
Error:  Illegal character found in source: (ISO Latin1) $e9, i.e., 'e'
> Object cafe
```

because identifiers may only use the letters A to Z, in upper and lower case, the digits 0 to 9 and the underscore character _. The same kind of error is produced if you try to use (say) the ^ character outside of quotation marks. The characters legal in unquoted Inform source code are:

⟨new-line⟩ ⟨form-feed⟩ ⟨space⟩ ⟨tab⟩
0123456789
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
()[]{}<>"',.:;?!+-*/%=&|~#@_

. . . . .

3. *Variables and arrays*

```
Variable must be defined before use: ⟨name⟩
'=' applied to undeclared variable
Local variable defined twice: ⟨name⟩
All 233 global variables already declared
No array size or initial values given
Array sizes must be known now, not externally defined
An array must have a positive number of entries
A 'string' array can have at most 256 entries
An array must have between 1 and 32767 entries
Entries in byte arrays and strings must be known constants
Missing ';' to end the initial array values before "[" or "]"
```

The limit of 233 global variables is absolute: a program even approaching this limit should probably be making more use of object properties to store its information. "Entries... must be known constants" is a restriction on what byte or string arrays may contain: basically, numbers or characters; defined constants (such as object names) may only be used if they have already been defined. This restriction does not apply to the more normally used word and table arrays.

. . . . .

4. *Routines and function calls*

```
No 'Main' routine has been defined
It is illegal to nest routines using '#['
A routine can have at most 15 local variables
Argument to system function missing
System function given with too many arguments
Only constants can be used as possible 'random' results
```

```
A function may be called with at most 7 arguments
Duplicate definition of label: ⟨name⟩
The following name is reserved by Inform for its own use as a
    routine name; you can use it as a routine name yourself
    (to override the standard definition) but cannot use it for
    anything else: ⟨name⟩
```

Note that the system function `random`, when it takes more than one argument, can only take constant arguments, as this enables the possibilities to be stored efficiently within the program. Thus `random(random(10), location)` will produce an error. To make a random choice between non-constant values, write a `switch` statement instead.

. . . . .

5. *Expressions and arithmetic*

```
Missing operator: inserting '+'
Evaluating this has no effect: ⟨operator⟩
'=' applied to ⟨operator⟩
Brackets mandatory to clarify order of: ⟨operator⟩
Missing operand for ⟨operator⟩
Missing operand after ⟨something⟩
Found '(' without matching ')'
No expression between brackets '(' and ')'
'or' not between values to the right of a condition
'has'/'hasnt' applied to illegal attribute number
Division of constant by zero
Signed arithmetic on compile-time constants overflowed the range
    -32768 to +32767: ⟨calculation⟩
Label name used as value: ⟨name⟩
System function name used as value: ⟨name⟩
No such constant as ⟨name⟩
The obsolete '#w$word' construct has been removed
```

"Operators" include not only addition `+`, multiplication `*` and so on, but also higher-level things like `-->` ("array entry") and `.` ("property value") and `::` ("superclass"). An example of an operator where "Evaluating this has no effect" is in the statement

```
34 * score;
```

where the multiplication is a waste of time, since nothing is done with the result. "= applied to ⟨operator⟩" means something like

```
(4 / fish) = 7;
```

which literally means "set 4/`fish` to 7" and gives the error "= applied to /".

"Brackets mandatory to clarify order" means that an expression is unclear as written, and this is often a caution that it would be wrong either way and needs to be reconsidered. For instance:

```
if (parent(axe) == location == Hall_of_Mists) ...
```

This looks as if it might mean "if these three values are all equal", but does not: the result of == is either `true` or `false`, so whichever comparison happens first, the other one compares against one of these values.

. . . . .

6. *Miscellaneous errors in statements*

```
'do' without matching 'until'
'default' without matching 'switch'
'else' without matching 'if'
'until' without matching 'do'
'break' can only be used in a loop or 'switch' block
At most 32 values can be given in a single 'switch' case
Multiple 'default' clauses defined in same 'switch'
'default' must be the last 'switch' case
'continue' can only be used in a loop block
A reserved word was used as a print specification: ⟨name⟩
No lines of text given for 'box' display
In Version 3 no status-line drawing routine can be given
The 'style' statement cannot be used for Version 3 games
```

For instance, `print (bold) X` gives the "reserved word in print specification" error because `bold` is a keyword that has some meaning already (the statement `style bold;` changes the type face to bold). Anyway, call such a printing routine something else.

. . . . .

7. *Object and class declarations*

```
Two textual short names given for only one object
The syntax '->' is only used as an alternative to 'Nearby'
Use of '->' (or 'Nearby') clashes with giving a parent
'->' (or 'Nearby') fails because there is no previous object
'-> -> ...' fails because no previous object is deep enough
Two commas ',' in a row in object/class definition
Object/class definition finishes with ','
```

```
⟨name⟩ is a name already in use (with type ⟨type⟩) and may not be
    used as a property name too
Property given twice in the same declaration, because the names
    '⟨name⟩' and '⟨name⟩' actually refer to the same property
Property given twice in the same declaration: ⟨name⟩
Property should be declared in 'with', not 'private': ⟨name⟩
Limit (of 32 values) exceeded for property ⟨name⟩
An additive property has inherited so many values that the list has
    overflowed the maximum 32 entries
Duplicate-number not known at compile time
The number of duplicates must be 1 to 10000
```

Note that "common properties" (those provided by the library, or those declared with `Property`) cannot be made `private`. All other properties are called "individual". The "number of duplicates" referred to is the number of duplicate instances to make for a new class, and it needs to be a number Inform can determine now, not later on in the source code (or in another module altogether). The limit 10,000 is arbitrary and imposed to help prevent accidents: in practice available memory is very unlikely to permit anything like this many instances.

. . . . .

8. *Grammar*

```
Two different verb definitions refer to ⟨name⟩
There is no previous grammar for the verb ⟨name⟩
There is no action routine called ⟨name⟩
No such grammar token as ⟨text⟩
'=' is only legal here as 'noun=Routine'
Not an action routine: ⟨name⟩
This is a fake action, not a real one: ⟨name⟩
Too many lines of grammar for verb: increase #define MAX_LINES_PER_VERB
'/' can only be used with Library 6/3 or later
'/' can only be applied to prepositions
The 'topic' token is only available if you are using Library
    6/3 or later
'reverse' actions can only be used with Library 6/3 or later
```

At present verbs are limited to 20 grammar lines each, though this would be easy to increase. (A grammar of this kind of length can probably be written more efficiently using general parsing routines, however.)

. . . . .

9. *Conditional compilation*

```
'Ifnot' without matching 'If...'
'Endif' without matching 'If...'
Second 'Ifnot' for the same 'If...' condition
End of file reached in code 'If...'d out
This condition can't be determined
```

"Condition can't be determined" only arises for `Iftrue` and `Iffalse`, which make numerical or logical tests: for instance,

```
Iftrue #strings_offset == $4a50;
```

can't be determined because, although both quantities are constants, the value of `#strings_offset` will not be known until compilation is finished. On the other hand, for example,

```
Iftrue #version_number > 5;
```

can be determined at any time, as the version number was set before compilation.

. . . . .

10. *Miscellaneous errors in directives*

```
You can't 'Replace' a system function already used
Must specify 0 to 3 local variables for 'Stub' routine
A 'Switches' directive must come before the first constant definition
All 62 properties already declared
'alias' incompatible with 'additive'
The serial number must be a 6-digit date in double-quotes
A definite value must be given as release number
A definite value must be given as version number
Grammar__Version must be given an explicit constant value
Once a fake action has been defined it is too late to change the
    grammar version. (If you are using the library, move any
    Fake_Action directives to a point after the inclusion of "Parser".)
The version number must be in the range 3 to 8
All 64 abbreviations already declared
All abbreviations must be declared together
It's not worth abbreviating ⟨text⟩
'Default' cannot be used in -M (Module) mode
'LowString' cannot be used in -M (Module) mode
```

. . . . .

11. *Linking and importing*

```
File isn't a module: ⟨name⟩
Link: action name clash with ⟨name⟩
Link: program and module give differing values of ⟨name⟩
Link: module (wrongly) declared this a variable: ⟨name⟩
Link: this attribute is undeclared within module: ⟨name⟩
Link: this property is undeclared within module: ⟨name⟩
Link: this was referred to as a constant, but isn't: ⟨name⟩
Link: ⟨type⟩ ⟨name⟩ in both program and module
Link: ⟨name⟩ has type ⟨type⟩ in program but type ⟨type⟩ in module
Link: failed because too many extra global variables needed
Link: module (wrongly) declared this a variable: ⟨name⟩
Link: this attribute is undeclared within module: ⟨name⟩
Link: this property is undeclared within module: ⟨name⟩
Link: this was referred to as a constant, but isn't: ⟨name⟩
Link: module and game use different character sets
Link: module and game both define non-standard character sets, but they
    disagree
Link: module compiled as Version ⟨number⟩ (so it can't link into this
    V⟨number⟩ game
'Import' cannot import things of this type: ⟨name⟩
'Import' can only be used in -M (Module) mode
```

Note that the errors beginning "Link:" are exactly those occurring during the process of linking a module into the current compilation. They mostly arise when the same name is used for one purpose in the current program, and a different one in the module.

. . . . .

12. *Assembly language*

```
Opcode specification should have form "VAR:102"
Unknown flag: options are B (branch), S (store),
    T (text), I (indirect addressing), F** (set this Flags 2 bit)
Only one '->' store destination can be given
Only one '?' branch destination can be given
No assembly instruction may have more than 8 operands
This opcode does not use indirect addressing
Indirect addressing can only be used on the first operand
Store destination (the last operand) is not a variable
Opcode unavailable in this Z-machine version: ⟨name⟩
Assembly mistake: syntax is ⟨syntax⟩
```

```
Routine contains no such label as ⟨name⟩
For this operand type, opcode number must be in range ⟨range⟩
```

. . . . .

13. *Deliberate "errors"*

Finally, error messages can also be produced from within the program (deliberately) using `Message`. It may be that a mysterious message is being caused by an included file written by someone other than yourself.

Warnings

1. *Questionable practices*

⟨type⟩ ⟨name⟩ `declared but not used`

For example, a `Global` directive was used to create a variable, which was then never used in the program.

```
'=' used as condition: '==' intended?
```

Although a line like

```
    if (x = 5) print "My name is Alan Partridge.";
```

is legal, it's probably a mistake: `x = 5` sets `x` to 5 and results in 5, so the condition is always true. Presumably it was a mistype for `x == 5` meaning "test `x` to see if it's equal to 5".

```
Unlike C, Inform uses ':' to divide parts of a 'for' loop
    specification: replacing ';' with ':'
```

Programmers used to the C language will now and then habitually type a `for` loop in the form

```
    for (i=0; i<10; i++) ...
```

but Inform needs colons, not semicolons: however, as it can see what was intended, it makes the correction automatically and issues only a warning.

```
Missing ','? Property data seems to contain the property name <name>
```

The following, part of an object declaration, is legal but unlikely:

```
with found_in MarbleHall
     short_name "conch shell", name "conch" "shell",
```

As written, the `found_in` property has a list of three values: `MarbleHall`, `short_name` and `"conch shell"`. `short_name` throws up the warning because Inform suspects that a comma was missed out and the programmer intended

```
with found_in MarbleHall,
     short_name "conch shell", name "conch" "shell",
```

```
This is not a declared Attribute: ⟨name⟩
```

Similarly, suppose that a game contains a `pen`. Then the following `give` statement is dubious but legal:

```
give MarbleDoorway pen;
```

The warning is caused because it's far more likely to be a misprint for

```
give MarbleDoorway open;
```

```
Without bracketing, the minus sign '-' is ambiguous
```

For example,

```
Array Doubtful --> 50 10 -20 56;
```

because Inform is not sure whether this contains three entries (the middle one being $10 - 20 = -10$), or four. It guesses four, but suggests brackets to clarify the situation.

```
Entry in '->' or 'string' array not in range 0 to 255
```

Byte `->` and `string` arrays can only hold numbers in the range 0 to 255. If a larger entry is supplied, only the remainder mod 256 is stored, and this warning is issued.

`This statement can never be reached`

There is no way that the statement being compiled can ever be executed when the game is played. Here is an obvious example:

```
return; print "Goodbye!";
```

where the `print` statement can never be reached, because a `return` must just have happened. Beginners often run into this example:

```
"You pick up the gauntlet."; score = score + 5; return;
```

Here the `score = score + 5` statement is never reached because the text, given on its own, means "print this, then print a new-line, then return from the current routine". The intended behaviour needs something like

```
score = score + 5; "You pick up the gauntlet.";
```

`Verb disagrees with previous verbs:` ⟨verb⟩

The `Extend only` directive is used to cleave off a set of synonymous English verbs and make them into a new Inform verb. For instance, ordinarily "take", "get", "carry" and "hold" are one single Inform verb, but this directive could split off "carry" and "get" from the other two. The warning would arise if one tried to split off "take" and "drop" together, which come from different original Inform verbs. (It's still conceivably usable, which is why it's a warning, not an error.)

`This does not set the final game's statusline`

An attempt to choose, e.g., `Statusline time` within a module, having no effect on the program into which the module will one day be linked. Futile. Finally a ragbag of unlikely and fairly self-explanatory contingencies:

```
This version of Inform is unable to produce the grammar table format
    requested (producing number 2 format instead)
Grammar line cut short: you can only have up to 6 tokens in any line
    (unless you're compiling with library 6/3 or later)
Version 3 limit of 4 values per property exceeded (use -v5 to get 32),
    so truncating property ⟨name⟩
The 'box' statement has no effect in a version 3 game
Module has a more advanced format than this release of the
    Inform 6 compiler knows about: it may not link in correctly
```

The last of these messages is to allow for different module formats to be introduced in future releases of Inform, but so far there has only ever been module format 1, so nobody has ever produced this error.

. . . . .

2. Obsolete usages

```
more modern to use 'Array', not 'Global'
use '->' instead of 'data'
use '->' instead of 'initial'
use '->' instead of 'initstr'
ignoring 'print_paddr': use 'print (string)' instead
ignoring 'print_addr': use 'print (address)' instead
ignoring 'print_char': use 'print (char)' instead
use 'word' as a constant dictionary address
'#a$Act' is now superseded by '##Act'
'#n$word' is now superseded by ''word''
'#r$Routine' can now be written just 'Routine'
all properties are now automatically 'long'
use the ^ character for the apostrophe in ⟨dictionary word⟩
```

These all occur if Inform compiles a syntax which was correct under Inform 5 (or earlier) but has now been withdrawn in favour of something better.

△△ No Inform library file (or any other file marked `System_file`) produces warning messages. It may contain many declared but unused routines, or may contain obsolete usages for the sake of backward compatibility.