A-code Documentation

Mike Arnautov

mipmip.org/acode

version: 2026-Aug-03

A-code documentation, 2025-May-06

A-code and where to get it
About A-code
A-code sources
Building games from source
Build types explained
Translating A-code into ANSI C
Creating game executables from derived C sources
Using *advbld* script (if you have *bash*)
Running A-code games
A-code language
A-code 12 reference
A-code parser
A-code texts
A-code vocabulary handling
A-code game internationalisation
Diving deeper
UNDO and REDO
Library mode interface
The context mechanism
Upward compatibility of games
Writing games in other languages
A-code debugging
Finally, some history
Differences between A-code styles
Language history, as I recall it.

A-code introduction

Dave Platt developed the A-code language (not to be confused with the Level 9 A-code system!) for writing his influential "Adventure 3" (a.k.a Adventure550) expansion of the original Crowther/Woods Adventure game. In doing so, he broke from the traditional format of Adventure and its expanded versions, relying on a custom executable processing a standard format data file.

Though only a few games have ever been written in A-code, it is of some historical interest because of the influence of Adventure3. Personally, I regret that the language did not catch on – it is easy to write, is very readable and has some unusual, useful features. To the best of my knowledge this is the complete list of A-code games:

- Adventure3 (a.k.a. Adv550 or PLAT0550) Dave Platt 1980
- Adventure C01 (a.k.a. Adv580 or GOET0580) Mike Goetz 1982?
- Adventure4+ (a.k.a. Adv660 or ARNA0660) Mike Arnautov 1984?
- Adv770 (a.k.a. ARNA0770) Mike Arnautov 2003

There are also a couple of A-code re-implementations of existing games:

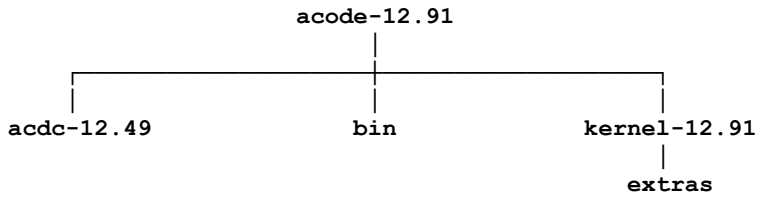
- Roger Firth's "Cloak of Darkness" – Cloak by Mike Arnautov, 1999
- Crowther's & Woods's Adventure – Adv350 by Mike Arnautov, 2020

This documentation describes the current version of A-code, which is still backward compatible with original A-code sources of all of the above games. Differences (both of implementation and of feature) between Platt's original version of the language and the current one are explained in the history sections of the document.

I am profoundly grateful to Joan CiberSheep for teaching me how to create pdf and epub versions of this documentation and for assistense in scripting the task!

Language sources and documentation

The A-code language sources tarball acode-12.91.tgz contains the following basic directory hierarchy:



All contents of this tarball are licenced under the GPL3 (or later) – see the LICENCE file in the top directory of the tarball.

- **acdc-12.49**
  C sources of the A-code to C translator acdc. This translator is used (and built) automatically by the bash *advbld* script, but it can also be used on its own. Once compiled and linked ("cc *.c -o acdc", which is done automatically by *advbld*), it will show usage if given -h (or --help) command line option. See acode-c-build.html for an explanation of uses of the translator if *advbld* is not available or cannot be used.
- **bin**
  Contains the following A-code bash script tools:
  - **advbld**
    An all-purpose bash script for building A-code game executables. See a separate page for a guide to this script's use. If, for whatever reason, you cannot or do not wish to use *advbld*, there is a separate page explaining how to create game executables from supplied C sources, and another page explaining how to convert A-code source to derived C source. **Please note** that in order to work, the script must be executed where it is in the supplied directories hierarchy, since it expects other parts of the hierarchy to be in the same relative positions, as supplied in the tarball. The top level directory *acode-12.91* can, however, be renamed as you please.

    - **lib.sh**
      A library of shell functions used by the *advbld* script.
- **doc**
  A-code documentation in HTML format.
- **kernel-12.91**
  A-code kernel C sources to compile and link with derived game C sources created by the *acdc* translator. The *extras* sub-directory contains bits and pieces required by specialist build modes (JavaScript/HTTP, library test and QT5).

Building A-code games from C sources

(A-code version 12.91)

Derived C sources

A-code games are, naturally enough, written in A-code, which is an IF writing language originally created by David Platt when developing his Adv550 superset of the original Adventure. However, the first step of converting A-code source files into an executable involves translating them into ANSI C. This is done by the *acdc* A-code to C translator and the process is covered in the section on building from A-code sources. Here I assume that you already have the derived C sources either downloaded as a part of the overall A-code tarball or produced by running the *acdc* translator yourself. The translator ANSI C sources are also included in the same tarball.

Building an executable from the C sources

A-code games can be built in 5 distinct modes: console, browser, single turn, HTML/JavaScript and library. This document explains the differences between these modes, their respective uses and the ways of building corresponding game versions from C sources.

If your system does not have the unistd.h header file, you should add -DNO_UNISTD to all compilation commands given below. An appendix lists and explains all compilation symbols that may be used in compiling A-code game C sources.

The *console* mode

The console mode is how the original Adventure was initially played. It takes input directly from the player and outputs any responses in plain text to a "computer terminal" – these days most likely a terminal emulator window. Some players still prefer this mode and it is also very handy for debugging.

While it still defaults to the dimensions of displays of ancient VDUs (24 lines and 80 fixed font characters per line), other dimensions can be specified on the invocation command line by means of the -s option. The kernel also provides the necessary hooks for a game to permit changes to these defaults while playing.

The console mode also offers a unique opportunity to experience an A-code game the way the original Adventure was played in 1970s/80s. The -o command line option allows the output speed to be set to the baud rate of 110 (a teletype), 300 (earliest VDUs), 600, 1200, 2400, 4800 and 9600. Younger players are hereby invited to marvel at the patience required to play at the lower speeds (as we did!).

For the full list of console mode command line options, please see the section describing command line invocation of A-code games.

If you have the *readline* and *ncurses* libraries installed and available for linking (which probably needs a readline development package), all you need to do is to use an ANSI C compiler to compile and link the C files, specifying the console mode:

```
cc -DCONSOLE *.c -lreadline -lncurses -o <game-name>
```

If you do not have those libraries installed, you can still build a console mode executable, which will lack the facility of recalling and editing previous commands. To do so, just tell the compiler not to use *readline*:

```
cc *.c -DCONSOLE -DNO_READLINE -o <game-name>
```

The *browser* mode

In the browser mode, A-code games do not interact with the player directly, but instead invoke a local browser and use that to render game's output and to obtain player's commands. In this mode, an A-code game acts as a very simple HTTP server.

Unless otherwise specified, the player's default browser is invoked, but another browser can be specified either on the invocation command line or by modifying the acode.config file created by the A-code kernel.

The browser build of an A-code game automatically includes the console build (but not the other way around!). Thus a browser-build game can be invoked in the console mode by adding -C to the invocation command line.

Building a browser mode executable is virtually the same as building a console mode one – you just drop the -DCONSOLE from the parameters given to the compiler. When a browser is being used to render game's output, there is no need for the readline library, so the basic browser mode build looks like this:

cc *.c -DNO_READLINE -o <game-name>

If you want the executable to offer command editing when run in the console mode, you will need to link in the two additional libraries:

cc *.c -lreadline -lncurses -o <game-name> *.c

The *single turn* mode

Originally developed for CGI operation, games built in this mode are only suitable for running in a cloud, via a suitable front-end script, e.g. a cgi-bin or a PHP one. In this mode, the game executable is supplied a single command as a parameter on the invocation command line, sends the text generated in response to standard output and exits. (See the the command line options section for details.)

The actual interaction with the player is carried out by the front end script, which repeatedly invokes the executable for successive game turns. The

secret sauce is, of course, the player-invisible save and restore of the current state of the game. This mechanism got later adapted for use by other game modes, automatically giving all A-code games a persistent state.

In this mode all text output is HTML-formatted, though this can be overridden if necessary. All such text is also prefixed by a single character, which provides information to the wrapper script and is not expected to be displayed to the player.

To build a single turn mode executable is very simple:

cc *.c -DTURN -o <game-name>

In this mode the executable is not responsible for acquiring player commands and thus there is no need to signal the absence of the readline library -- this is assumed automatically.

The *HTML/JavaScript* mode

Thanks to the magic of emscripten, A-code games can be built as pure (and purely local, with no network dependencies) HTML/JavaScript page, usable by any HTML5 compliant browser. In this mode, A-code games run entirely within the player's browser and use browser's own sand-boxed file system for saving and restoring games.

For building games in this mode you will need *emscripten* installed as well as its dependencies (*clang* and *cmake*). Actually building a game involves two steps. Firstly the derived C sources (plus the kernel ones) are converted to JavaScript via this one-line command:

emcc -Os -s ASM_JS=1 -s WASM=0 -s ENVIRONMENT=web -DJS adv*.c -Wno-parentheses-equality -lidbfs.js -s "EXTRA_EXPORTED_RUNTIME_METHODS=['cwrap']" --memory-init-file 0 -o acode.js -s EXPORTED_FUNCTIONS="['_advturn']"

(That's the command line for Emscripten 1.39.6 -- experience suggests that later versions may require a change in some of the arguments.)

Secondly, the resulting *acode.js* file has to be merged into the *acode.html* template, which can be found in the tools directory to be found in the relevant A-code tarballs. This merging is achieved by (a) replacing all occurrences of the string %NAME% in the template by the name of the game being built and (b) replacing the line consisting of the single token %JAVASCRIPT% with the contents of the JavaScript file generated by *emcc*.

The resulting HTML file can be called anything you like, but it seems a good idea to replace 'acode' in its name with the actual name of the game.

The *library* mode

In some cases it is not feasible or appropriate for an A-code game to drive its own command/response loop. App frameworks generally expect to do so themselves. This situation is handled by compiling derived C sources in the library mode:

```
cc -c *.c -DADVLIB
```

Compiled this way, instead of having a main() routine, the game has an advturn() one, which expects player input to be supplied via its arguments and returns a pointer to a text buffer containing the resulting text. As in the single turn mode, the returned text is prefixed with a single character denoting the nature of the text.

A separate section explains the details of the advturn() interface. However, for testing/debugging purposes I had to develop a simple C program, which uses the library mode and can be run in any terminal emulator. Its source, *libtest.c* can be found in the tools directory of an A-code tarball.

Appendix 1: Deprecated text data handling

By default, since version 12.01 of A-code, all text data is preloaded by the **acdc** translator into the game C-source, whereas previous versions by

default stored text data in a separate .dat file. The current arrangement is the simplest and most sensible one for most machines these days, but a few years ago I still saw some not entirely obsolete machines being defeated by the size of adv770 executable with preloaded data. Hence other options are also on offer, even if you don't have access to game A-code sources.

- **Creating the text data file**
  Even though game C sources no longer come with the .dat text data file, this file can get automatically constructed by the game executable. To do this, first build the executable in one of the three ways described below (for preloading, paging or reading of the data file), and then run the resulting executable in the directory containing the C sources. Failing to find the .dat file, but finding adv6.h instead, the game will construct the .dat file from the .h one. From then on the game will run normally, getting its text data from the .dat file, which should be kept in the same place as the executable (but see the section on game invocation command options for an alternative).

- **Loading text data on game startup**
  If -DMEMORY is added to any of the C compilation commands suggested above, the text data file will be preloaded into a dynamically allocated buffer on game startup. This used to be the default arrangement in A-code versions 10 and 11.

- **Reading data from the text file**
  If your memory is constrained, but disk I/O is reasonably quick (which it would be unless you are running the game from a floppy disk), you can use the -DFILE compilation flag. This causes the game to read from the data file with no paging of its own, though one hopes that *some* paging will be done by the OS. As a guess, such machines will have no GUI browsers, so -DCONSOLE should also be added to compilation flags.

- **Keeping recent text data in memory**
  If -DSWAP is added to any of the C compilation commands suggested above, all access to the text data file will be through an internal paging system of 32 1KB buffers, paging data in and out on the first-in-first-out basis. The number of swap buffers can be modified by defining the

SWAP symbol to have a particular value, e.g. -DSWAP=40. The number of buffers will be coerced into the range from 16 to 128 inclusive.

This method is only useful for the oldest and slowest machines. Again, it probably makes sense to add -DCONSOLE as well, since a machine with such a limited amount of memory is unlikely to sport a GUI-based browser.

Appendix 2: Kernel compilation symbols

Compilation symbols listed below can be added to the compilation command line as -D<symbol> when building A-code games. There is a number of other symbols being defined and used by kernel source files, which you should leave well alone.

The main mode symbols have been already referenced above.

CONSOLE Specifies console-only mode

CGI Specifies single turn mode

HTTP Specifies combined browser/console mode – this is the default

JS Specifies JavaScript/HTML mode

ADVLIB Specifies library mode

Four additional symbols can be used to control what system routines the game executable needs from the system.

NO_UNISTD Signals absence of the unistd.h header file. Use it in all modes if you do not have the unistd library.

NO_READLINE Suppresses use of the readline library (only relevant in the console and browser/console modes – ignored otherwise).

NO_SLOW Suppresses the ability to slow down game's output (only relevant in the console and browser/console modes – ignored otherwise).

PAUSE Prevents immediate exit on game completion (useful when running in console mode in a window which automatically closes on game's exit).

Three further symbols which are only required in kernel development work and/or in special builds.

IOS Signals iOS build.

DEBUG Enables various debug messages from the kernel.

DIRECT Prevents the HTTP server being daemonised in the browser mode – makes debugging easier!

Finally, there are the deprecated symbols specifying ways of dealing with a separate text data file, if there is one. Note that these symbols override whatever text data arrangement was specified in running **acdc** to convert A-code sources into C.

MEMORY Text data file to be loaded into memory on startup.

FILE Text to be retrieved as required directly from the data file.

SWAP[=<page_buffers>] Text is to be paged in executables 1KB page buffers. The default is 32 such buffers, but another number can be specified between 16 and 128 inclusive.

Creating ANSI C sources from the A-code source

(A-code acdc version 12.91)

A-code source

Logically, an A-code source is a single file, usually with the .acd suffix. The suffix is optional; if present, it can be omitted when nominating to **acdc** the A-code file to be processed.

In practice, A-code source can be split into a number of files, which are incorporated in the "main" one by means of the A-code *include* major directive. Again, such include files are conventionally suffixed with .acd, which suffix may be omitted in *include* statements.

A-code framework

To build a working executable out of A-code source, you need A-code kernel files, available as a part of the acode system source tarball https://mipmip.org/acode/acode-12.91.html. This tarball contains C sources of the latest releases of the **acdc** translator and of the A-code kernel, as well as some useful bash scripts. See that page for details.

If you are using Linux, Unix, OSX/MacOS or Microsoft's LSW, the simplest thing to do is to use the **advbld** bash script supplied as a part of the acode source tarball – please see the README file contained therein. However, if you are using some other platform, you will probably need to build and use **acdc** yourself. The rest of this document explains how to do that.

Building the acdc translator

To build the **acdc** executable you need only an ANSI C compiler. Simply compile and link the relevant C sources – no libraries or special compilation or linking options required:

```
cc *.c -o acdc
```

Using the acdc translator

By default, any resulting C sources (adv.c and adv.h) are generated alongside the A-code source file being translated, unless the sub-directory C (that's capital C!) us found there, in which case that's where the sources go.

The **acdc** translator needs no command line arguments. If some are supplied, they can be given in any order. If the game source-file is omitted, it is prompted for. All other command line listed below are optional – default behaviour covers all usual cases.

<sourcefile>
: The name or pathname of the main A-code source file; if omitted, it is prompted for. Any include statements are taken to be relative to the directory in which the main source file is located.

-plain abbreviable to **-p**
: Causes the game text data not to be encrypted.

-xref abbreviable to **-x**
: Requests that a cross-reference file of the A-code source be created. For technical reasons (meaning I am being lazy), this file is called **game.xrf**. It is not sorted and can be processed further with the **sortref** Perl utility, supplied as a part of the acode package.

-no-warnings abbreviable to **-nw**
: Suppresses warnings about unused symbols in the A-code source.

-quiet abbreviable to **-q**
: Suppresses most of the standard info messages generated by acdc in translating the A-code source.

-debug abbreviable to **-d**
: Causes A-code source to be added as comment lines to the translated C source files. Also adds trace message showing individual A-code code chunks being entered during play. Also causes the DEBUG symbol to be defined in the kernel source.

-help abbreviable to **-h**
: Lists available command line arguments.

By default, the game's text data is preloaded into the executable. These days only very old and/or small machines are unhappy with the size of the resulting executable. However, if the default behaviour is for any reason not the suitable one, there are three further deprecated command line options that can be used.

-file-memory abbreviable to **-fm**
: Requests a separate .dat text data file to be created and to be read in full into the game's memory on startup. Useful only if the OS objects to large executables.

-file-read abbreviable to **-fr**
: Requests a separate .dat text data file to be created and to be accessed by the game by direct file reads as required. Only useful for *really* small, slow machines.

-file-page [<npage>] abbreviable to **-fp**
: Requests a separate .dat text data file to be created and to be accessed by the game using its own internal paging mechanism. The optional <npage> argument defaults to 32 and specifies the number of 1KB paging buffers. Only useful for *really* small, slow machines.

Adding kernel source files

The A-code kernel consists of source files **adv00.c**, **adv01.h** and **adv0.h**. All three can be found in the A-code source tarball. Copy them alongside the C sources created by **acdc**, and you will have the complete C source required to build an executable.

Building an executable

Building a simple adventure executable from the derived C sources is not complicated: just compile and link the lot together with any ANSI C compiler, adding -DNO_READLINE to the command line. The resulting executable will default to using a local browser for its display (except for DOS builds). But it will lack command history and editing in the console mode.

If you cannot use a browser or prefer not to, you can also run such default build from a console window by adding -C (or /C on DOS and Windows) to the invocation command line.

If that's good enough for you, fine, but if you hit problems, or want a more sophisticated version, read the guide to building A-code games from intermediate C sources.

Using the *advbld* script

- Requirements
- Assumptions
- Basic usage
- Advanced usage
- Brief options summary
- Options explained

My main tool for building A-code games is **advbld** – a *bash* script for building various modes of A-code games from their A-code sources (or from derived C ones). It can be found in the *bin* sub-directory of the directory tree supplied in the stand-alone A-code source tarball A-code source tarball and also bundled with A-code sources of individual games.

Requirements

The script should be usable on any Unix-like system which supports the GNU *bash* shell (version 3 or later) – i.e. Linux, various versions of Unix, MacOS and these days, perhaps even Windows under LSW.

For its simplest functionality, the script requires nothing more than the presence of an ANSI C compiler (e.g. *gcc* or *clang*) invokable as *cc*. The JavaScript/HTML build also requires Emscripten, while a Qt5 build needs Qt5WebKit and Qt5WebKitWidgets libraries, usually not installed by default with Qt5.

If GNU *readline* and *ncurses* libraries are present, these will be automatically used if necessary. Without them, the console build of games will lack command editing functionality.

Assumptions

The script makes some important assumptions.

- The script needs to use the *acdc* translator and kernel files. Directories containing these are assumed to be found alongside the script's directory, as supplied in the tarball. It follows that the script should be invoked where it is, in its *bin/* directory. You can invoke it by its pathname, of you can add that *bin/* directory to the command search path.

- The *acdc* translator demands that all A-code source files have the **.acd** suffix. If a file name is specified (be it on the command line or by the INCLUDE directive in the code) without this suffix, it will be automatically appended before any attempt to open the file.

- Unless the script is given the pathname of the game's source, it needs to be able to make a reasonable guess as to what source file should be used. To do so, it assumes that game source file(s) live in a directory of the same name as the name of the game, possibly tagged with the game version number. So for example, the main source file of adv770 is assumed to be **adv770.acd** to be found in the directory called **adv770** or e.g. **adv770-2.19**.

- By convention, sources found in a directory without a version number tag are deemed to be "unstable" – i.e. under development, whereas directories tagged by a version number are assumed to hold stable game versions. This distinction is only of use to game developers.

Basic use of *advbld*

The most straightforward, though not always the most convenient, way to use the script is to supply it with the pathname of a game's source file.

```
$ advbld ~/games/adv770/adv770.acd
```

or alternatively

```
$ cd ~/games/adv770
$ advbld adv770.acd
```

The *.acd* suffix can be dropped – if absent, it will be automatically appended by *acdc*.

Perhaps more conveniently, the script can infer the name of the main source file from the name of the current directory. So for example

```
$ cd ~/games/adv770-2.19
$ advbld
```

works too. The script will take the name of the current directory, stripping off the version number if necessary, and then look for a file named **adv770.acd** or **main.acd** or **game.acd**.

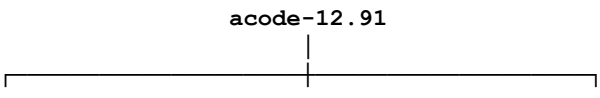
If a game is specified just by name and there is no game file of the corresponding name in the current directory, it will be searched for – the directory specified by the environment variable **$ADVDIR**, if this is set, and then in the likely places relative to the location of the *advbld* script itself. The upshot is that in most cases you can just type e.g. "advbld adv770" and let the script do the finding. For more details, see the next section on advanced uses of the script.

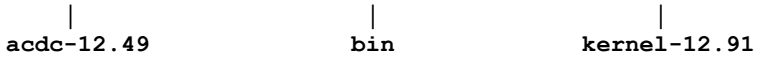
In all cases, the executable will be built in the same directory in which the game source is located.

By default, the script builds the combined browser/console executable. By default the executable will invoke your default browser and use that as the game interface. Alternatively, if invoked with the **-C** command line option, it will run simply as a console mode program. See the next section for other build modes.

Advanced use of *advbld*

As supplied, the A-code sources tarball has the following directory hierarchy:





The *advbld* script can be found in the *bin* sub-directory of the directory tree supplied in the A-code tools tarball. You can change the name of the top directory of that tree, but the rest of it must stay as supplied in order for *advbld* to work. The tarball does not contain an executable of the *acdc* translator. It will be build by the script automatically the first time it is required using an ANSI C compiler invokable as *cc*.

The *acdc* and *kernel* sub-directories can be versioned (e.g. *acdc-12.49*) or non-versioned (e.g. just *acdc*). By convention, directories without version numbers contain "unstable" code – still under development. Versioned directories contain stable code of the appropriate version.

By default, the *advbld* script will use either non-versioned directories for acdc, kernel and whatever game is being built, or failing that, the directory of the right name with the highest version number available. This default behaviour can be modified by invocation options used. For example, one can supply a game name with its version number, to force the particular version of the game code to be built. Or one can use the -s option, to ignore non-stable code.

When searching for a game specified just by its name, which is not found in either in the current directory or in an appropriately named sub-directory of $ADVDIR (if this is set), the script will look first alongside its **bin** directory or alongside its parent (acode-12.91 in the above diagram).

If you wish to build a game in the HTML/JavaScript mode, you will also need to install and configure Emscripten (https://kripken.github.io/emscripten-site/).

A QT5 build requires Qt5 installed and dev versions of these libraries: Qt5Core, Qt5Widgets, Qt5 Gui, Qt5WebKit and Qt5WebKitWidgets.

If you have Perl installed, the script can generate sorted cross-reference lists of game A-code sources.

Usage: **advbld [options] [game]**

To build a game, *advbld* needs: game source(s), the acdc translator and the A-code kernel files. All of these are expected to be found in sub-directories of the parent directory of the one containing the *advbld* script.

The directory containing game source(s) is assumed to bear the name of the game itself, possibly suffixed with a version number (e.g. adv770 or adv770-2.19).

The *acdc* translator is expected to be found in the directory called 'acdc'. If this is absent, directories named 'acdc-<version>' (e.g. 'acdc-12.36') are examined and the one with the highest version number is used. (The non-versioned directory is assumed to contain unstable sources and is ignored if a stable build is requested.)

Similarly, the kernel files are assumed to be either in the directory called 'kernel', or (if this is absent or if a stable build is requested) in the highest version number directory named 'kernel-<version>'.

By default, *advbld* assumes the name of your current directory to be the name of the game to be built. If a game name is given, it will be assumed to live in a directory of that name alongside the directory containing the script itself. This is why game sources available from my mipmip.org are supplied in a directory tree of the same name as the A-code tools one. The A-code tools and any game sources tarballs are guaranteed to co-exist peacefully within that top level directory with no clashes.

For explanation of build modes available, please see a separate page.

Here's a quick summary of *advbld* options (see below for more detailed explanations).

Main build options

-B default; build a combined console/browser (HTTP) executable
-C build a console-only executable (no HTTP)
-J build an HTML/JavaScript version
-L build a test executable using the library mode

-Q build a QT5 executable

-T build a single turn (cloud) mode executable

-W include opt/debug.acd if it exists. This option is compatible with all of the above build types.

Script display options

-v show more progress info

-q less progress info

-x echo commands being executed

-h show available options

The rest of options are of use only to an A-code game developer.

Versions of acdc, kernel and game are deemed "unstable" if the relevant directories lack a version number. The script defaults to using unstable versions, if present, and the latest (highest version number) versions otherwise.

Version-related options

-s	use the latest **stable** versions of acdc and kernel (and game)
-u	insist on using **unstable** versions of acdc and kernel (and game)
-a <version>	use the specified version of acdc
-k <version>	use the specified version of kernel

Options passed through to the C compiler

-g	create a gdb-instrumented executable
-gg	like -g plus gcc macro storing
-D<symbol>	add a symbol to the C compilation command
--m32	force building 32 bit executable, if possible

Options modifying *acdc* behaviour

-p	(plain) don't encrypt game data
-d	acdc's -debug – adds A-code lines as comments in derived C

-c	translate A-code to C but do not build executable
-b	don't translate A-code to C but do build executable
-w	show acdc warnings, which are suppressed by default
-X	generate A-code source cross-reference lists

Now for some more details... Please note that the script may also understand some deprecated options not listed here. Those will simply disappear in due time.

Main build options

-B (default)

Browser/Console build

This option instructs *advbld* to build a browser-capable executable. By default such an executable uses a local browser for interacting with the players. The executable itself acts as a very simple HTTP server, passing player commands to the game, and game's responses to the player. The default browser is invoked for this purpose (in a manner appropriate to supported platforms), but a different browser can be nominated on the command line following this option, or by editing the .acode/acode.conf file. This build can be also used to play in a console (terminal) window by invoking the executable with the -C option.

-C

Console mode build

The game executable expects to run in a console (or terminal) window. It offers the unique opportunity of adventuring the way it was on old slow output devices. The output speed in baud can be specified by adding "-o <baud>" to the invocation command line, where "<baud> is a number such as 300 (old teletypes), 1200 (DECwriters) etc...

-L

Library mode build

By default, A-code games are in control of the command/response loop, but this is not possible in some cases (e.g on IOs). This "library" option converts kernel's *main()* into *advturn()*, which returns every time the player is prompted for input. It is up to the calling program to

obtain player command and to supply it in the next call to *advturn()*. Please see the section on details of the call interface.

-J

JavaScript/HTML build

Building a JavaScript version requires presence of *emscripten* (see the emscripten home page for installation instructions). This option causes *advbld* to create a self-contained HTML/JavaScript page, which will run the game in any HTML5-compliant browser.

-Q

QT5 build

This build puts a QT5 wrapper around the library mode build. It requires dev versions of the following QT libraries: Qt5Core, Qt5Widgets, Qt5Gui, Qt5WebKit and Qt5WebKitWidgets.

-T

Single turn (cloud) build

The resulting executable is suitable for cloud use with a suitable front-end. It restores game in progress, takes player command as command line arguments, executes a single game turn, outputs the result on standard output as HTML-formatted text, saves game in progress and exits. A new game can be forced by using the -n command line keyword instead of player command. A save game can be loaded by starting the executable with -l <saved_game>.

-W

Debug (wizard mode) build

If the file opt/debug.acd exists, it gets copied to the current directory before building the game. The copy is deleted after the build is complete. This mechanism presupposes debug.acd being conditionally included by the game's A-code source.

Script display options

-v

Requests more progress info

Adds *acdc*'s report to the information being displayed.

-q

Requests less progress info

Only the final success (or error report) is produced.

-x
Echo commands being executed by the *advbld* script

-h
Show available options of the *advbld* script

Version-related options

-s
Force use of the latest stable acdc and kernel (and game) versions
Stable kernel versions live in directories named **kernel-<version>**. Similarly for the *acdc* translator and individual games. This option overrides the default behaviour of preferring unstable versions of acdc, kernel and game code. Forces the use of the highest version number ones. Only of use to game developers.

-u
Forces use of the *un*stable acdc and kernel (and game) versions
Only of use to game developers.

-a <version>
You may have older versions of *acdc* in directories called **acdc-<version>**. This option instructs *advbld* to use the specified *acdc* version, instead of searching for the most recent one.

-k <version>
You may have older versions of the kernel in directories called **kernel-<version>**. This option instructs *advbld* to use the specified kernel version, instead of searching for the most recent one.

Options passed through to the C compiler

-g
Create a gdb-instrumented executable

-gg
As -g but forces use of gcc and makes macro definitions available to the gdb debugger. Note that this is **not** a GNU-style long option. It is in fact interpreted as -g -g.

-D<symbol>
Define an additional compilation symbol
NB: you can (but need not) add a space between -D and the symbol name.

--m32

Force 32 build

Only of use on 64 bit machines.

Options modifying *acdc* behaviour

-p

(Plain) don't encrypt game data

By default game data (mostly text) is encrypted in order to make it harder to cheat by examining core dumps. The disadvantage of this is that executables are less compressible.

-d

acdc's -debug – adds A-code lines as comments in derived C

In the absence of an A-code debugger, this is a moderately convenient way of debugging games on the A-code source level.

-c

Translate A-code to C but do not build executable

Just invokes *acdc* to translate game's A-code to C.

-b

Don't translate A-code to C but do build executable

This option omits running *acdc* – useful if temporarily tweaking translated C sources while debugging.

-w

Show *acdc* warnings

Warnings about unused symbols in the A-code source are suppressed by default.

-X

Generate A-code cross-reference

Requests the *acdc* translator to construct sorted cross-reference lists of the A-code source

A-code build types

A-code games can be built in a number of different ways, such as e.g. console only. This document explains all these build types and their uses. For details on actually building games in various modes, please see a separate page explaining how to build A-code games from derived ANSI C sources.

The ***console*** build
The ***browser*** build
The ***library*** build
The ***JavaScript*** build
The ***QT5*** build
The ***single turn*** build

The *console* build

The console build replicates how the original Adventure was initially played. It takes input directly from the player and outputs any responses in plain text to a "computer terminal" – these days most likely a terminal emulator window. Some players still prefer this build type and it is also very handy for debugging.

While it still defaults to the dimensions of displays of ancient VDUs (24 lines and 80 fixed font characters per line), other dimensions can be specified on the invocation command line by means of the -s option. The kernel also provides the necessary hooks for a game to permit changes to these defaults while playing.

The console build also offers a unique opportunity to experience an A-code game the way the original Adventure was played in 1970s/80s. The -o command line option allows the output speed to be set to the baud rate of 110 (a teletype), 300 (earliest VDUs), 600, 1200, 2400, 4800 and 9600.

Younger players are hereby invited to marvel at the patience required to play at the lower speeds (as we did!).

For the full list of console build command line options, please see the section describing command line invocation of A-code games.

The *browser/console* build

By default, A-code games built in this way do not interact with the player directly, but instead invoke a local browser and use that to render game's output and to obtain player's commands. In this mode, an A-code game acts as a very simple HTTP server. No access to network is involved – there's just local socket-based communication between the game and the browser.

Unless otherwise specified, the player's default browser is invoked, but another browser can be specified either on the invocation command line or by modifying the acode.config file created by the A-code kernel.

The browser build of an A-code game automatically includes the console build (but not the other way around!). Thus a browser-build game can be invoked in the console mode by adding -C to the invocation command line. All console build command line options apply when -C is specified.

The *library* build

In some cases it is not feasible or appropriate for an A-code game to drive its own command/response loop. App frameworks generally expect to do so themselves. This situation is handled by compiling derived C sources in the library mode – used, for example, in Brian Ball's iOS port of Adv770 and in HTML/JavaScript builds of A-code games.

A simple such wrapper (*libtest.c*) is provided with kernel sources and should be compiled and linked with derived C-sources and with kernel sources to produce a library build executable.

The *HTML/JavaScript* build

Thanks to the magic of emscripten, A-code games can be built as pure (and purely local, with no network dependencies) HTML/JavaScript page, usable by any HTML5 compliant browser. If built this way, A-code games run entirely within the player's browser with no access to the network, and use browser's own sand-boxed file system for saving and restoring games.

The *QT5* build

The QT5 build uses the library mode of A-code and wraps it in a QT5 GUI front-end.

The *single turn* build

Originally developed for CGI operation, single turn game builds are only suitable for running in a cloud, via a suitable front-end script, e.g. a cgi-bin or a PHP one. In this mode, the game executable is supplied a single command as a parameter on the invocation command line, sends the text generated in response to standard output and exits. (See the the command line options section for details.)

The actual interaction with the player is carried out by the front-end script, which repeatedly invokes the executable for successive game turns. The secret sauce is, of course, the player-invisible save and restore of the current state of the game. This mechanism got later adapted for use by other game modes, automatically giving all A-code games a persistent state.

In this build all text output is HTML-formatted by default, though this can be overridden if necessary, by adding -C to the invocation command line. All output text is also prefixed by a single character, which provides information to the wrapper script and is not expected to be displayed to the player.

A-code game invocation options

(A-code version 12.91)

This document describes command line options available when running an A-code game. Game behaviour is generally regulated by the internally documented acode.conf file, which can be found in the **.acode** directory (just **acode** on MS platforms), automatically created in player's home directory. Where command line options refer to particular features covered by the configuration file, they override the configuration file settings.

Conventions:

- Angle brackets <string> denote a symbolic string to be replaced by something appropriate. E.g. <filename> would be replaced by the name of a file (with no surrounding angel brackets).
- Square brackets [ ] denote something optional. So e.g. -l[<logfilename>] means that the name of the log file may be omitted.
- Braces { } denote a list of permissible values, separated by vertical bars |. E.g. -b[{0|1|2}] means the -b may be optionally (square brackets!) followed by one of the three digits zero,one or two.
- All options are shown with a short (single dash) prefix, but double dashes are also accepted.
- Where a value can be specified with a command line option, the syntax shown is that of the value abutting directly to the option specification letter. However, an equality sign = can be placed between the two, so that -b0 is equivalent to -b=0.

The following command line options are valid for both the browser and the console display modes:

-n

Force a new game. By default, if a previous game session got somehow forcibly interrupted (e.g. by the game process being killed for whatever reason), the interrupted session is automatically resumed

when the game is restarted. The -n option overrides this behaviour and forces the interrupted session to be forgotten.

[-r]<dumpfile>

Restore game from dump. Ignored if the game does not support game dump files being specified on the command line. The -r optional in that any command line argument which does not begin with a dash will be interpreted as the name of the dumpfile to restore on invocation.

-l[<logfile>]

Log the game. Specifies the file into which a session log is to be written. The log is human-readable, but has some additional information useful for debugging. If the nominated logfile is in fact a directory, the default log name is used (game name, sifficed with .log). If no logfile name or pathname is specified, the log is created alongside game's saved files and the default log name is used. already exists, it gets appended to.

-B<browser>

Browser mode executables only. Use a non-default browser for player interaction. Browsers can be specified by their pathname, or by their name – in the latter case the name is searched for in directories given by the PATH variable.

-C

For browser-mode executables, force console display – i.e. do not use the browser interface. For single turn mode executables this option enforces plain text, non-formatted output, instead of default HTML output.

--

For single turn executable, treat the rest of the command line as the player's game command.

-b[{0|1|2}]

Set or invert the blank line setting. If set to zero blank lines are inserted before and after each prompt. If the value is 1, blank lines around '?' prompts are suppressed, resulting in a more compact display.

If the value is 2, then ALL blank lines are suppressed, for super-compact, but less readable output. If no value is specified, the new setting is 0 or 1, inverting the A-code default for this game. In old-style A-code (Adv550), which does not distinguish between replies to queries and general commands, this only affects presence/absence of a blank line after the prompt line, and never before it.

-u{0|1|none}

Set the initial state of undo-history collection. Ignored if the game does not support undo. If the value is zero, the default undo status is OFF. If the value is one, the default undo status is ON. The "none" state implies OFF and disallows undo functionality being subsequently switched on from within the game. The default state is ON for games which define the verb UNDO, and "none" otherwise.

-v

Show the game, kernel and acdc version numbers and exit.

-h

Print command line usage summary appropriate to the mode of the executable's build.

In addition, some options are only meaningful in the console display mode, and are simply ignored in other modes.

-j[{0|1}]

Set text mode to wrap (0) or justify (1). If no value is specified, invert the default A-code setting for this game. In wrap mode, text is simply broken into lines according to the screen width (see the -s option below). With justification turned on, each line is right-justified. All of this presupposes a fixed font being in use. For variable font devices, which tend to do their own wrapping, the default screen width should be set to zero, meaning "infinite", and the margin should be specified as zero too. This option is ignored by games written in the "old style" A-code (i.e. by adv550).

-s<W>.<H>[.<M>]

Set screen size (width in fixed font characters, height in lines, and margin in fixed font blanks). The default screen dimension is 80x24-1, the margin being set to 1 character. The -s option allows a different screen size (and optionally margin) to be specified. Screen width of zero means "infinite" width. Note that the line length cannot be set to less than 16 characters and the minimal number of lines per screen is 5.

-o<baudrate>

Set the output speed as specified by the argument. Meaningful only in the "dumb" console mode and only if C sources compiled without the NO_SLOW symbol. Baud rate is specified in bits per second, and taking into the account control and parity bits, the output speed in characters per second is simply the baud rate divided by 10. The game coerces the specified baud rate to the nearest lower standard value (one of 110, 300, 600, 1200, 2400, 4800 and 9600), except that anything below 110 (the speed of a teletype) is also treated as 110. The default value is 300 - the speed of a DECwriter. Note, however, that under DOS and Windows any baud rate above 600 results in no slowdown at all.

-p[{0|1}]

Pause on exit. Requests that after printing the final exit message the game should prompt the player for a <CR>, before exiting. This feature is intended for players who wish to play console version of the game, in a window which closes as soon as the game exits.

The functionality of the **-s**, **-u** and **-j** options is also provided via kernel hooks (see procedure special() in the kernel source file adv00.c), so that the game may -- at author's discretion -- offer the player commands for toggling the justification switch, switching on and off the change history, and altering screen size and margin.

Finally, if the game was build to have a separate data file (only useful for DOS builds, nowadays), a non-default location of that file may be specified via the -d option.

-d<dbsdir>

If using a data file, specify its directory. Ignored if the game is built as a simple executable with no associated data file. By default, the game data file is assumed to live in the same directory as the executable. The -d option allows a separate location to be specified. The program will attempt to work out the separator which should follow the directory name, but if in doubt as to the appropriate one for the given platform, the dbs name will be simply concatenated with the supplied pathname -- hence if it doesn't work, try adding the trailing separator to the pathname.

Any unknown or suppressed keywords are quietly ignored.

A-code library mode interface

Normally, A-code games drive their own main loop, but various app frameworks insist on taking this functionality over. To cater for implementations based on such frameworks, A-code kernel has a *library mode*. When compiled in this mode, instead of having main(), it has a function called advturn(), which takes as its sole character string argument a a request (e.g. a player's command), and returns as its value a pointer to the game's response.

```
char *advturn(char *command)
```

Examples of such builds are the emscripten-based HTML/JavaScript build of Adv770 and Brian Ball's iOS build of the same game. The advbld script can be used to build test library mode versions of game executables. It uses the *libtest.c* wrapper provided with the kernel sources.

At present, there is one important restriction on A-code games to be built using the library mode: they cannot use the QUERY directive. Reasons for this restriction, as well as a workaround, can be found in the section describing the A-code CONTEXT mechanism.

The advturn function takes a character string as its single argument and returns a pointer to a character string. This pointer is maintained by the kernel and should not be manipulated by the calling code.

Once the game is running, the argument string contains a game command obtained from the player and the returned pointer points to the game's response to that command. However, there are also other special values that the argument string can have:

"_INFO_"
: Requests game's info. The returned string contains game's name, version and date.

"_LIST_"

Requests the list of saved games separated with '|' (vertical bar) used as a separator. If there is a game-in-progress save, it heads the list under the name of the game prefixed with a dot (e.g. .adv770).

"_START_[{TEXT_|HTML_}]"
Requests a new game to be started. The optional "TEXT_" or "HTML _" force use of plain text or HTML output, as appropriate. The default is HTML.

"_RESUME_[{TEXT_|HTML_}]"
Requests the game-in-progress (if any) to be resumed. Text or HTML output can be nominated in the same manner as for _START_.

"_LOAD_[{TEXT_|HTML_}]<save_name>"
Requests the nominated saved game to be restarted. Here too, "TEXT_" or "HTML_" can be used to force the desired output format, e.g. "_LOAD_HTML_mygame".

Except for the above special cases, the argument string is interpreted as the player's next command to be processed by the game. The output string contains the full game's response, prefixed with a single character indicating the type of the response:

- 'f' means this is the final response – the game is over
- 'q' means the response is a query of some sort (not necessarily a yes/no one)
- 't' signals just ordinary text, which should be followed by a prompt before requesting player's next command
- 'n' signals null response (this happens if the command string is null). The rest of the response is unchanged from the previous non-null command.

The default game output format is HTML, except for console mode builds where plain text is enforced. Paragraphs of plain text output are **not** split into separate lines – any wrapping, if required, has to be done by the calling code.

The simple *libtest.c* wrapper program supplied with the A-code kernel sources provides an example of using the library mode.

OBSOLETE: The CONTEXT mechanism or how to avoid use of QUERY

The way A-code is it was implemented until version 12.92, had one substantial disadvantage: single turn builds (JS/HTML, cloud and library ones) could not use the QUERY directive at all and permitted only a singe INPUT use in the main loop. This restriction is now gone, so the below discussion is obsolete. The CONTEXT mechanism is now deprecated and will be removed at some future time..

The two input directives (INPUT and QUERY) expect execution to proceed from the the point where they have been called, which is fine if the game executable takes care of its own main loop. However, in the single turn and library modes, this is not the case. Input is obtained by other controlling software (e.g. by a PHP script, in the single turn mode) and there is no mechanism to return the game to the point at which an input directive is used. The problem is aggravated by the fact that it may make perfect sense to have multiple calls for input in a single pass through the main loop (e.g. get general command, if this command is SAVE without a name to save the game under, get the name; if there is already such a saved game, query whether it should be overwritten).

In Platt's original A-code implementation as a virtual machine this would not have been a problem. One simply saved the state of the VM before returning control to the external software. Once the game code is re-invoked and supplied with the next command, the VM state is restored and execution proceeds as normal. Things are nowhere near as simple when a game is simply an executable (or a library) built from C sources.

There are possible solutions, of course. The game could be split into two processes: one in charge of player communication and the other running the game's world. Indeed, that's how the browser mode build works, but this arrangement is not suitable for running in a third-party cloud.

The CONTEXT mechanism was my solution to this problem when implementing the original CGI-based implementation of Adv770 to be used

by game's beta-testers. The idea is a simple one. The game can acquire query responses as a part of the main loop – it just needs to know that a question has been asked and specifically *what* question has been asked. Then having acquired player's input it can deal with it appropriately.

At first glance, no kernel involvement is required for this to work, but there are reasons why it does have to be aware of this mechanism. Otherwise anomalies would arise in e.g. orphan command word handling and the game persistence mechanism. Thus variable named CONTEXT is deemed to be special by the kernel. If it is set to zero, the next expected input is just a general game command. If set to non-zero, however, an answer to a particular query is expected by the game.

Here is a very simple example. First a trivial A-code program using QUERY:

```
style 12
verb quit
init
repeat
# (Some house-keeping code)
repeat
  input               # No initialisation
  query "Really?_"    # The underscore is a forced space
    ifkey quit
      say "Bye..."
      stop
    else
      say "OK."
    fin
  else
    say "If not, not."
  fin
  say "_"             # Add a blank line
```

And here is one possible equivalent using the CONTEXT variable instead of the QUERY directive:

```
style 12
var context
verb quit
verb yes              # Any other query answer is deemed to be no
init                  # CONTEXT automatically initialised to zero
repeat
  ifturn
    and
  ifflag context, prompted
    proceed
  fin
```

```
# (Some house-keeping code here)
repeat
  ifeq context, 0     # A general command is required
    proceed
  fin
  save command        # Preserve player's original command
  input               # Get query response
  ifkey yes
    restore command   # Restore the original command
    ifkey quit        # And act on it if appropriate
      say "Bye..."
      stop
    fin
    say "OK."
  else
    say "If not, not."
  fin
  set context, 0      # Ensure next loop is just a general command one
  quip "_"            # Add a blank line
repeat
  input
  set context, 1      # Signal query processing is needed
  quip "f:Really?_"
```

Note the addition to the first, house-keeping REPEAT section. In single turn builds it ensures that the housekeeping code is not executed twice, whenever the player is prompted for input.

Probably the best way of getting to grips with the context mechanism is to examine differences between A-code sources of Dave Platt's non-CGI and my CGI versions of Platt's Adv550.

A brief summary of A-code, version 12

A-code is the adventure-writing language developed by Dave Platt for the adv550 superset of the original Adventure, and subsequently developed by Mike Arnautov, in working on the adv660 superset and later on the adv770 one.

Introduction

A-code is not intended to be a general purpose programming language and its strengths lie in being tailored specifically for programming purely text-based interactive fiction games. Each game created with the A-code system is a stand-alone package, requiring no separate game engine/interpreter. As an added bonus, with a little care it is possible to ensure that any game saves are upward compatible and thus can be used even after the game has been modified/extended (a feature of particular use in beta-testing!).

The current implementation of the language has only been used for writing extensions of the original Adventure, and hence its parser is limited to dealing with commands which can be reduced to (a series of) verb/noun commands. (See the section on the current A-code parser.)

On the positive side, that implementation is sufficiently UTF8 compliant to allows games to be written in some languages and character sets other than English/Latin.

Technically, A-code is a Polish notation (i.e. "prefix", or "operator [argument [...]]") language, so its statements take the form of, e.g. "set door, oiled".

The original version by Dave Platt was organised as a "munger", translating A-code source into tokenised pseudo-binary, and an interpreter, interpreting the pseudo-binary at run time. This was an economical arrangement, but as versions grew in size, its performance on a multi-user machine gradually became less satisfactory. (We are talking mid-80s here! :-))

This is why the current implementation of A-code takes a different route. It consists of an A-code to C translator (".acd -> .c" or just acdc!) and a C kernel which gets compiled and loaded with the translated code. The overall size of the program is larger, but the performance is very much better.

Now that computer speeds have increased dramatically, it would probably make sense to replace this translation-based implementation with a virtual machine, akin to Platt's original implementation.

A-code style and version numbering

The current version of A-code departs in some significant ways from Platt's original version of the language, but maintains full backward compatibility with it. This is achieved by an optional explicit declaration via the STYLE directive of the major version number of A-code being use by game source. The significance of the major A-code version in save game compatibility.

- Style 1 is Dave Platt's original A-code of Adv550
- Style 2 is Make Goetz's slightly different A-code of Adv580
- Styles 3 to 9 were development stages of Mike Arnautov's Adv660
- Style 10 is the A-code version of the finalised code of Adv660
- Style 11 was for a while the A-code of Mike Arnautov's Adv770, until save game compatibility got broken by some technical developments.
- Style 12 is the current A-code of Adv770

If not explicitly declared, the default style is assumed to be the currently highest one supported by the A-code engine being used.

A note on conventions and notation

All A-code documentation uses the following notation:

- A-code statements are case insensitive. This documentation uses upper case except when providing code snippets – that is just to make such statements to stand out against other text.

- A-code lexical elements (tokens) are separated by spaces and/or commas. By convention, the operator (the first token) is separated from its arguments simply by a space (or several spaces), while its arguments are separated by commas followed by a space (or spaces). E.g. "APPORT BOTTLE, YLEM".
- Angled brackets (< and >) are used to denote a generic name or value, to be substituted for as appropriate. Thus, e.g., "SET <varname>, <value>" is a generic form of statements such as, e.g., "SET COUNT, 1".
- Square brackets ([and]) are used to indicate optional arguments. For example "LOCAL <varname1> [, <varname2> [...]]
" indicates that more than one local variable can be declared in a single LOCAL statement.
- Ellipsis ([...] or just ...) as in the above example, is used to show that the preceding element can be repeated.
- Curly brackets ({ and }) are used to denote mandatory items permitting some alternatives, such as "SET <entity>, {<entity2>|<state>|<constant>}".
- The word "entity" is used as a collective noun for any declared elements, which are not mere synonyms for numerical constants.

Source file naming.

All A-code source file must have the mandatory suffix **.acd**. However, in using A-code tools or when using the A-code INCLUDE directive to include another source file, this suffix may be omitted – it will be automatically added by the software.

A-code program structure.

At the highest conceptual level, an A-code program falls into four distinct parts:

- Header statements giving game information, (optional)

- Declarations of game-specific constants, entities and procedures (also optional)
- Code to be executed once, on game's invocation (mandatory, but may be empty)
- The main code loop, usually prompting the player for commands, and processing those commands (mandatory)

Thus the simplest A-code program looks like this:

```
init    # There is actually no initial code
repeat  # Main loop...
   stop # ... consisting simply of the instruction to terminate the game
```

And here is the obligatory Hello World code:

```
init
   say "Hello world!"
repeat
   stop
```

In practice it is often convenient to have more than one INIT and REPEAT sections, which are executed in the order of their declaration.

Major (or declarative) directives fall into three categories: header ones, translator directives (or pragmas) and entity declarations. With a single exception, they can occur in any order. All major A-code directives (translator instructions and declarators) must start in column one. All other lines of A-code source must be offset from the beginning of the line by one or more tabs or spaces.

Other source lines are either parts of declaration, or lines of code. The latter have the form of a minor directive (a.k.a. opcode), possibly followed by some arguments. Code line tokens are separated by commas and/or spaces. The convention of using comma-space separators is not mandatory.

A-code lexical items

A-code lexical items fall into three categories:

1. Major directives (or opcodes)

2. Minor directives (or opcodes)
3. Ad hoc modifiers to minor directives

Major directives come in three flavours:

- Header statements describing game name, author, date etc...
- Pragmas, which affect the way the game source is treated by any software processing it (e.g. an instruction to include a separate source file in place of the INCLUDE pragma).
- Declarations of game elements, such as location, objects, procedures etc. Forward declarations are allowed! That is, a game entity may be referenced before it is declared.

Minor directives are, with one exception (the LOCAL opcode, declaring local variables of a procedure) instructions for performing some action (e.g. SET STATUS, 1) and are used to write A-code procedures.

Types of declared elements

Elements manipulated by A-code are variables, objects, places, texts, words, procedures, flags, states and constants. All declared elements are global in scope, except for local variables and procedure arguments, both of which are local to the relevant procedure.

Some elements (specifically flags, states and constants) are merely synonyms for integer values. Procedures (or procs) are a category of their own. All other elements are entities in their own rights and are referred to collectively as entities in this document.

In its current implementation, A-code makes no distinction between vocabulary verbs and nouns, allowing e.g. "cage" to be treated as either, depending on context. However, the basic form of a player command is verb optionally followed by a noun. Thus in the context of parsing player commands "verb" and "noun" will be used, referring respectively to the first and second words of player command.

Any source line, which is not a part of a text or a place.object description is considered to be terminated by a hash (#) sign. Thus # can be used to provide comments, including in-line comments. (This convention holds for A-code version 10 and higher -- earlier versions had different convention for comments.)

Declared element names are subject to these rules:

1. All names are case-independent.
2. Names may not contain blanks or other "white space" characters (such as tab, end-of-line etc.).
3. Unless UTF8 encoding is being used, all names must start with an alpha character, a dot or a question mark.
4. Unless UTF8 encoding is being used, The rest of the name characters may be any of: an alpha character, a digit, dot (.), dash (-), underscore (_) shriek (!), single quote ('), ampersand (&) and forward slash (/).

All declared (or automatically declared) elements other than states, flags or constants are automatically assigned a "reference number", or *refno* for short. You do not need to worry about specific refnos, but it is useful to understand that (a) they exist and (b) how they are assigned.

Refnos are assigned separately for each type of entity, consecutively in the order in which elements are declared in the source code. Thus e.g. objects will have consecutive refno value, even if their declaration is interspersed with declarations of other entities. Refnos can be thought of as "addresses" of relevant entities and, indeed, can be used as such.

Unlike other entities, texts can be declared nameless. Such texts can be accessed and manipulated via refno offsets from the nearest preceding named text. (Note, however, that this feature is deprecated in favour of using text switches.) In-line texts are also permitted. See the A-code texts description later in this document.

Entities other than vocabulary words also have attributes other than their assigned refnos. They have a *value* (a short integer) and a bit-screen of binary *flags*. Objects, places and texts have further attributes appropriate to

their type. These will be covered when describing declarations of such entities.

All entity values are automatically initialised to zero and all flags are automatically set to false on program start-up.

Places and vocabulary words can have (and generally do) have procedures associated with them via the AT and ACTION major directives (see below).

Flags

Flags are binary yes/no properties (bits in entity bitscreen), referred to by symbolic names. A flag's value is simply a bit-screen offset. There are three separate categories of flags. One is defined for objects, one for places and one for variables. Each entity has its own instance of the appropriate kind of a flag set. There are no limits on the number of flags in a flag set.

Even within a given set, each individual flag can have several different names, allowing for the greater legibility of the source. Since alternative names are usually required for variable flags, several different variable flag sets can be defined (as opposed to only one for each the object and the place flag sets). The actual variable flag set is simply the biggest of these, with individual flags having synonymous names as defined in different set definitions. Don't worry if this makes little sense - it will eventually.

Constants

A-code constants come in several flavours, but fall into two simple types: numerical (integer values only) and symbolic. Where a integer value has been given a symbolic name, this can be used interchangeably with the actual value it names.

The various constant flavours are:

- Numerical integer values (signed short integers)

- Symbolic names of such values, declared as CONSTANT for ease of reading game sources.

- Some entities (places, objects, variables and texts) each have a state (a.k.a. value) associated with them. These can be named by using STATE declarations. The only difference between a state and a symbolic constant is that the former cannot be negative.

- Some entities (namely objects, places and variables), also carry a bitscreen of binary flags. Symbolic names should be given to individual flags via FLAGS declaration – these are effectively just symbolic names for bitscreen offsets (necessarily non-negative).

  Flag names differ from state names in some important ways. Firstly, there can be only one set of flags associated with locations, whereas a number of multiple flag sets can be declared for objects and variables. Secondly, the distinction between object, place and variable flag sets is used as part of syntax checks. Thirdly, the system uses flag declarations to work out sizes of bitscreen required respectively by objects, places and variables.

- Finally, there are compound constants. These are deprecated. See the deprecated features section for a brief explanation.

Variables

Variables are symbolically named entities, each with a state and an instance of the variable flag set. There are some mandatory variables (e.g. HERE, see below), but any number of additional variables can be declared by the programmer. Words entered on the command line are available through the special mandatory variables ARG1 and ARG2 (and ARG3, currently used for handling of EXCEPT, but available for a future parser use of holding command instrument). This generally restricts A-code to simple verb/noun commands, even though the parser makes this less then obvious, permitting apparently complex commands such as e.g. DROP ALL BUT LAMP AND

ROD THEN EXIT (see the parser section for an explanation of the way player commands get parsed).

It is important to understand that the ARGn variables are unlike any others. They have the appropriate player command word associated with them regardless of the actual variable value (unless the value is -1, meaning "no word"). This makes error handling possible, because variable value can be sat by the parser to an appropriate error code, while embedding the variable in a text to be displayed, actually embeds the corresponding command word.

A special form of variable declaration is an **array**. In A-code this has a very limited, simple meaning: declaring an array of n elements declares a variable of the given name, followed by n-1 anonymous variables. (See below on handling of anonymous entities).

An automatic indirection occurs for some A-code directives when manipulating variables into which "addresses" (refnos) of other entities have been loaded. Generally speaking this happens for non-arithmetical manipulation of variables.

Texts

Texts are named, nameless, or in-line entities representing a piece of text. Any amount of text can be associated with a text entity. Generally multi-line texts are considered to be unformatted, line wrapping and filling being done by the A-code kernel at run time. However, A-code texts are more than just textual strings. The section on A-code texts gives their full description.

Procedures

A-code procedures are unusual in that there can be any number of procedures sharing the same name. Such procedure groups are executed in the order of their occurrence in the source (i.e. in the order of their refnos).

Procedures come in 5 varieties:

1. Initialising ones, declared as INIT. These form a procedure group, which gets executed on game startup.
2. Main loop, declared as REPEAT. These form a procedure group executed repeatedly after the INIT procedure group execution is completed.
3. "Free-standing" declared as PROC (or PROCEDURE). These procedures are all named. Procedure arguments (if any) are specified on the declaration line after the procedure's name. If two or more form an identically named procedure group, they must all have the same number of arguments.
4. Associated with a place and declared as AT (as in "at such-and-such-place"). Their names are in fact names of relevant locations. No arguments are allowed.
5. Associated with a vocabulary word and declared as ACTION. The name of such a procedure is the vocabulary word in question (or equivalently, any of its synonyms). Arguments are allowed and, perhaps surprisingly, procedures in the same procedure group will have different arguments and even different number of arguments. See the minor directive ACTION below for details.

Places

Places are symbolically named entities, each with a state, an instance of the place flag set, up to three different unnamed texts (brief, long and detailed descriptions) and a "hook" for unnamed chunks of A-code to be associated with the place. By default symbolic place names are not available in the player vocabulary, but each such name can be specifically forced into the vocabulary by prefixing the name in the place declaration with a plus sign (+). Any number of synonyms can be defined for a place name, though of course, there is no point doing this unless such names are forced into the player vocabulary. All properties of texts (described above) apply to place descriptions.

Objects

Objects are also individually named (with any number of synonymous names) and once again, have to their name a state and an instance of the object flag set each, plus up to three different descriptions (inventory, long and detailed), but no A-code. Object descriptions also obey the rules common to any kind of text. Object names (and synonyms) do get entered into the player vocabulary by default, but it is possible to exclude any specific object name by prefixing the object name in its declaration with a minus sign (-).

Words (a.k.a. verbs, a.k.a. nouns)

Words have symbolic names (with synonyms) and each is associated with one or more chunks of A-code. By default, verb names are added to the vocabulary, but again, any specific name can be excluded. It is sometimes useful to have "dummy" verbs for internal house-keeping, but not available to the player. This is again achieved by prefixing the verb name with a -, in the verb declaration.

As noted above, some of the entity names are automatically or selectively added to the player vocabulary.

To summarise, vocabulary words consist of: all explicitly defined words/verbs/nouns and their synonyms, all object names and their synonyms not explicitly excluded and some those place names which are explicitly included. The vocabulary section gives a full description of A-code vocabulary.

Major A-code directives

Major directives are either declarations or translator instructions. They must start in the first column of a line – that's how they are recognised in the first place, since everything else must be indented. The indentation depth is

arbitrary – the convention of 9 leading spaces and subsequent indents of three spaces at a time is not mandatory.

Pragmas

Firstly, there are two major directives, which are in fact translator instructions:

INCLUDE <file>
: read commands from indicated file until end of file, then revert to reading from the previous file. Include files may be nested down to 10 levels.

INCLUDE? <file>
: conditional include – includes the file if it exists. If it does not exist, this directive is simply ignored. Given the fact that A-code permits forward declarations and supports procedure groups, this pragma can be used for optional addition of new commands or modification of existing commands – a feature used extensively by Adv770 to provide the optional Wizard (debug) mode.

Game info, a.k.a header directives

Secondly, there are six game information directives, all of them optional.

NAME <game-name>
: names the game (a.g. adv770). If not specified, the game's name is taken to be the name of the file given to the acdc translator, less the .acd suffix.

VERSION <version>
: specifies game's version. Traditionally this has the format of two numbers (major and minor versions) separated by a dot (.), however, this format is not enforced. If omitted, the current year is used as the game's version.

DATE <date>
gives a free-format specification of the date of this version of the game.

AUTHOR <author-name>
name the game's author (also free format).

STYLE <style>
specifies the A-code style, i.e. the major version number of A-code in which the game is written. If used, directive must precede any non-header directives. If omitted, style is set to the major version of the *acdc* translator used to convert the game's A-code source into ANSI C.

Style 1 is reserved for Dave Platt's original code of Adv550. Style 2 is similarly reserved to Mike Goetz's Adv580 (an expansion of Adv550). Styles 3 to 9 are not in use. Style 10 is reserved for Adv660.

UTF8
signals use of the UTF8 character encoding in the game's A-code source. This directive is only required if the game uses UTF8 characters in its entity names or vocabulary words.

(There are also a couple of header directives (GAMEID and DBNAME) which are obsolete and only supported in style 10, i.e. in Adv660.)

Declarations

The remaining major directives are all declarators.

Other than the flags, states and constants, all declared entities are associated with a "reference number" (refno) and are internally referenced by A-code **only** by this number. The reference numbers of entities of the same kind (e.g. objects, places, texts, variables...) are guaranteed to be lumped together into an unbroken numerical interval, with their reference numbers incrementing by one in the order in which the entities are declared.

To reiterate: within each declarative category (e.g. objects, places, texts) entities are stored sequentially in the order of declaration and hence can be referenced via refnos by numerical offsets from other entities.

It is not necessary for the game author to be aware of the reference numbers of declared entities, but it is helpful to be aware of this arrangement in general terms.

Reference numbers are used to achieve indirection in A-code. They can be loaded into variables, which are then recognised by most opcodes as "indirectors" so that the opcode is actually applied to the entity whose reference number the variable stores. This does not apply to arithmetical opcodes, making it possible to do "reference number arithmetic". This allows, for example, referencing nameless texts by offsets from other named texts, and has other uses too. It is **not** necessary to know the "reference number" of an entity, in order to load it into a variable.

FLAGS {VARIABLE|OBJECT|PLACE}

starts the declaration of a flag set of the appropriate kind (variable, object or place). It is followed by any number of indented lines, each declaring a symbolically named flag in the set, optionally followed on the same line by synonymous names. Only one object and place set declarations are permitted, but any number of variable set declarations are allowed. (They are equivalent to a single declaration with flags declared on corresponding lines being synonymous with each other).

STATE [value] statename [...]

is used to declare symbolic names for entity states. It makes "statename" synonymous with the value preceding it, or with zero, if the value is absent. A STATE directive can be followed by any number of indented lines of the format " [value] statename", which declare further state names to be synonymous with the corresponding value, if given, or with the value of "statename" defined on the preceding line plus 1. Note that the STATE directive deliberately does not specify which entities the defined statenames relate to, allowing for partial state sequences to be shared by different entities. "Value" can be a number, a previously declared constant, a previously declared entity

(in which case the "reference number" of the entity is used) or several such, combined into a simple expressions with plus (+) and/or minus(-) signs, without any separating blanks.

CONSTANT [value] constname [...]

This directive is in fact exactly synonymous with the STATE directive. It is used simply to indicate that the symbolic constants being defined are general purpose constant and not entity state names. The two are completely interchangeable.

TEXT [textname]

declares a piece of text, which may be associated with a text name or be "anonymous" (available only by offset from some named text). The TEXT directive is followed by one or more lines of text, which must **not** start in column 1 (anything starting in column 1 is taken to be a major directive -- it follows, that all leading spaces are ignored). A-code texts are very rich entities. Please see the section on A-code texts for details.

FRAGMENT textname

Identical to the TEXT directive, except that the text is not terminated by a new line. I.e. it is a text fragment.

PLACE [+]placename [...]

declares a location. By default, "placename" is not entered into the player vocabulary, unless prefixed with a '+' (which does not count as a part of the name). Synonymous names can be declared on the same line.

Each location declaration is optionally followed by a piece of text (following all the text rules described above), comprising up to two different descriptions – the brief one and the full one – and terminated by a major directive. Insofar as these do description components occur – they occur in that particular order. The brief description (if any) immediately follows the PLACE line and is terminated by a text line starting with a '%' or a major directive. The full description (if any)

starts with a line beginning with '%' and is terminated by the next major directive.

If the full description is missing and the brief one is present, the full one defaults to the brief one. It is possible for a location to have no description at all. The '%' delimiter is not itself considered to be a part of the description. If a text switch is encountered in a description being displayed, the current state of the location is used as the switch qualifier. Thus for example

```
PLACE +TUBE
   You're in lava tube at top of chimney.
  %You're at the top of a narrow chimney in the rock. A cylindrical tube
   composed of hardened lava leads south and northwest[/, from where
   comes some ][/silvery /day/orange ][/light, providing a dim
   illumination to this place].
```

All special trickery described in the A-code texts section applies to place descriptions as well.

OBJECT [-]objname [[=]objname ...]

declares an object. By default, "objname" is entered into the player vocabulary, unless prefixed with a '-' (which does not count as a part of the name). Synonymous names can be declared on the same line. These can be prefixed with a '=', indicating their re-mapping to the last synonym not so prefixed, which, in fact may be a string (see the example below), which is not itself entered into the game's vovabulary.

Each object declaration is optionally followed by a piece of text (following all the text rules described above), comprising up to three different descriptions – the inventory one, the full one and the detailed one - and terminated by a major directive. Insofar as these three description components occur – they occur in that particular order. The inventory description (if any) immediately follows the OBJECT line and is terminated by a text line starting with a '%', '&' or a major directive. The full description (if any) starts with a line beginning with '%' and is terminated by a line beginning with a '&' or by the next major directive. The detailed description (if any) starts with a line beginning with a '&' and is terminated by the next major directive. E.g.

```
OBJECT "nest with eggs in it", =NEST, =EGGS
   Nest with golden eggs
  %There is a large nest here, full of golden eggs!
```

```
&The nest holding the eggs is lined with some sort of bird down - might
 be goose, but I am not sure. The eggs themselves gleam dully in the
 light with the unmistakable gleam of pure gold.
```

If the full description is missing and the inventory one is present, the full one defaults to the inventory one. If the detailed description is missing, it defaults to the full one, if that is present, or to the inventory one, if the full one is absent and the inventory one present.

It is possible for an object to have no description at all. The '%' and '&' delimiters are not themselves considered to be a part of the description. If a text switch is encountered in a description being displayed, the current state of the object is used as the switch qualifier.

All special trickery described in the A-code texts section applies to place descriptions as well.

NOISE word [...]

NOISE declares words to be ignored when parsing player's input.

VERB [{-||}]verbname [synonyms]

Declares a word "verbname" (optionally with synonyms), which is (by default) entered into the vocabulary available to the player. The optional minus sign prefixing the principal verb word stops the verb being available in the vocabulary. Such dummy verbs have two separete uses. I only the principal name is given, the word is useful for range checking within the vocabulary and bracket groups of verbs of a similar kind (e.g. movement directions). Synonyms to such excluded words can be used in replacing words that have to be know to the kernel (such as AND, THEN or AGAIN) with non-English alternatives. See the internationalisation section for details.

VERB is currently synonymous with NOUN and WORD. Adjectives and prepositions are not currently supported. Distinction between different word types are up to the game's code.

ACTION verb [objname]

The major directive ACTION is used to associate chunks of A-code (procedures) with individual verbs. More than one procedure can be assigned to the same verb and they are executed in the order of their

declaration. If the optional object name is given, that particular action procedure is skipped, unless the object in question features in player's command.

VARIABLE variable [...]

defines one or more variables. Note that variables defined on one line like this are **not** synonymous. By convention, related variables tend to be defined in a single VARIABLE directive.

ARRAY arrayname constant

defines a consecutive block of variables of the size given by the constant. The first of these is named by the array name, the rest are anonymous – accessible only via refno offsets from the array's base

PROC procname [arg ...]

defines a chunk of A-code (a set of executable code) called "procname". The procname can be followed by a list of arguments, which are passed by value when the procedure is called. The subsequent lines contain the code, terminated by the next major directive.

AT placename

defines code to be executed when the player is at the indicated place - the following lines contain the actual code. More than one AT procedure may be defined for a particular place – they are executed in the order in which they are declared in the A-code source.

INITIAL

defines once-only code to be executed at initialisation time. Multiple INITIAL commands may be used and are executed in the order encountered.

REPEAT

defines the main action-processing code that is executed during each player input. After the INITIAL code has been executed, the REPEAT statements are executed. Once the last REPEAT statement is executed, the program loops back and starts again with the first.

Each of the above "major directives" must appear in column 1. A major directive embraces all following lines up to but not including the next major directive statement (i.e., all lines in which column 1 is blank) or the end of the source file in which it occurs.

Special entities and flags

Not all entities or flags need to be declared explicitly. Some are "automatic" and some are "optional"

"Automatic" means that the entity is declared automatically. Automatic entities may be explicitly declared by the program, but if so, must be of the correct type. If an entity of that name is declared as some other type, this is treated as a compilation/translation error.

"Optional" means that the entity may but need not be declared by the program. If declared, it has a special meaning. An entity of that name but of a different type is treated as a compilation/translation error.

INHAND automatic location

Use: contains all objects currently carried by the player; contents are maintained by the kernel but can be also accessed and modified by the game's code.

STATUS – automatic variable

Use:
set by the kernel to indicate the number of words in the player's latest command.
Can be set by the program to various values in order to pass information to the kernel.
Possible values:
BADSYNTAX – automatic state
Use: set by the kernel if the current player command cannot be parsed.
NO.MATCH – automatic state
Use: set by the program to indicate that no abbreviation or approximate matching is to be performed on the next player command.

NO.AMATCH – automatic state
Use: set by the program to indicate that no approximate matching is to be performed on the next player command.
value **-1**
Use: set by the kernel to signal game restore on start-up.
Flags:
FULL.DISPLAY – optional flag
TERSE.DISPLAY – optional flag
MOVED – optional flag
Use: set by the kernel whenever player's location changes.
JUGGLED – optional flag
Use: set by the kernel if player's inventory has changed.
PLS.CLARIFY – optional flag
Use: set by the game's code in order to trigger orphan processing of next player input; cleared by the kernel as a part of orphan processing. (See the description of the A-code parser for an explanation of orphan processing of commands.)

HERE – automatic variable
Use: set by the kernel to the refno of the player's current location.

THERE – automatic variable
Use: set by the kernel on a change to player's location to the refno of the location prior to the change.

ARG1 – automatic variable
Use: set by the kernel to the refno of the player command's verb
Possible values:
BADWORD – automatic state
Use: set by the kernel if the word is not understood.
AMBIGWORD – automatic state
Use: set by the kernel if the player's word can be an abbreviation of more than one vocabulary word.
AMBIGTYPO – automatic state
Use: set by the kernel if the player's word can be matched as a typo in more than one way
SCENEWORD – automatic state

Use: set by the kernel if the player's word is not found in the vocabulary but matches a word (in excess of three characters) in the latest location description or in a text displayed since the last move.

ARG2 – automatic variable
Use: set by the kernel to the refno of the player's noun (if any) or zero otherwise.
States: (as ARG1)

ARG3 – automatic variable
Use: At present used solely for EXCEPT processing. If player's command is of the form "<verb> ALL EXCEPT <list>", ARG3 may be set by the kernel to BADWORD, AMBIGWORD or AMBIGTYPO, if appropriate.

CONTEXT – obsolete and deprecated optional variable
No longer required since the QUERY directive (and multiple INPUT directives) work in all modes from A-code 12-92 onwards.
Prior use: avoid use of the QUERY minor directive in game builds which require single-turn operation (see the section on A-code game build modes).
Value: If non-zero, indicates the current command to be a response to a query, the value (as set by the program) indicating the nature of the query.
Flags:
PROMPTED – automatic flag
Use: set by the kernel if the player has been prompted.

ENTNAME – optional variable
Use: when processed by the SAY directive, shows the symbolic name of the entity to which ENTNAME is set to point.
Value: a pointer to a named entity set by game's code.
Use of ENTNAME has the two side-effects:

- It stops the dot character ('.') acting as a player command separator, permitting reference to entity names containing this character.
- It a player command word is not found in the games vocabulary, game entity names are also searched by the parser; only exact matches are accepted – no abbreviations and no typo matching.

The variable is intended for game debugging in the wizard mode. See the section on debugging A-code games.

TYPO – optional text

Use: enables single-typo matching of player commands against the games vocabulary.

The TYPO text must contain a switch of 4 (or a multiple of four) components. It is used by the kernel to explain a typo match to the player. Regardless of the number of components, it must be declared as

```
FRAGMENT CYCLE TYPO
```

The first three components in each foursome must contain one word holder (#). The four switch components (in any of the switch quads if there are more than 4 components) are used by the kernel as follows:

1. Shows original player word that got typo-corrected
2. Shows the typo-corrected version for what the player gave
3. Shows the full non-abbreviated matched word if typo-corrected version is an abbreviation.
4. States that the corrected version is assumed

Here is an example from Adv770:

```
FRAGMENT cycle TYPO
    [Sorry, the word "#" is not familiar to me./ I'll just assume you
    meant "#"/, i.e. "#"/./
    Regretfully, I don't have "#" in my dictionary./ But I do know
    "#"/, as in "#"/, so I'll assume you meant that./
    The word "#" is not one I know./ Let's assume you meant "#"/,
    meaning "#"/./
    I wonder what "#" might be.../ I guess it could be mistyped
    "#"/, i.e. "#"/, so I'll assume that was what you meant.]
```

UNDO and **REDO** optional verbs.

Use:

Their presence activates the A-code undo/redo facility, enabling players to undo a specified number of last commands and, if necessary, undo some or all of this undoing. This is described in the section dealing with undo/redo.

UNDO.STATUS – optional variable

Use: set by the kernel to indicate the number of commands undone or re-done, which need not be the number requested by the player. If the variable is declared, the below flags are declared automatically:
UNDO.INFO – automatic flag
Use: Can be used by game code to note that player has been advised of rules for using undo; set to off by default and ignored by the kernel.
UNDO.TRIM – automatic flag
Use: maintained by the kernel: if set, the level of undo/redo was found to be excessive and had to be trimmed to match the existing undo history.
UNDO.INV – automatic flag
Use: maintained by the kernel: if set, the player's inventory has changed as the result of undo/redo.
UNDO.BAD – automatic flag
Use: maintained by the kernel: if set, undo/redo failed.

DWARVEN – optional variable
Use: If set to non-zero, all text output by the game is shifted circularly one position up in the alphabet and all player input is shifted one position down. If set to zero, any portion of game text delimited by % signs is output shifted one position up.

PROMPT – reserved optional variable
Use: none at present. Reserved for specifying non-standard prompt.

SCHIZOID an automatic object flag
Use: Indicates that the object is to be treated as present in its current location, and *also* in the immediately succeeding one (i.e. the one with the refno one higher).

Minor A-code directives (a.k.a. opcodes)

The actual A-code code consist of minor directives (or opcodes) followed by their arguments (if any). A line of code cannot have more than a single opcode line.

The following is a list of the available opcodes and a quickie description of what they do. As noted previously, in some cases if an argument is a variable, it is automatically de-referenced – i.e. the operation indicated by the opcode is applied not to the argument but to the entity referenced by the argument. Any such opcode arguments are denoted by a trailing asterisk in the following summaries.

Local variables

LOCAL varname [varname...]

Declares correspondingly named variables local to a procedure. It must immediately follow either the major directive declaring the procedure in question, or another LOCAL declaration. Local names pre-empt any identically named global variables. Local variables are dynamically initialised to zero value and zero bitscreen whenever the procedure is entered. They do not exist outside the procedure.

Local variables are not a part of the data set saved/restored by saving or restoring a game image. Thus they can be used to preserve some values over a game restore, instead of using the (now obsolete) EXEC 6 and EXEC 7.

Conditionals

All conditional structures have the basic form of

conditional, some code, FIN
conditional, some code, ELSE, some code FIN

where the immediately following block of code is executed if the conditional is true, and the block following ELSE (if present) is executed if it is false.

In order to avoid deeply nested indentation in source code, A-code also features OTHERWISE, analogous to C's "else if" or Perl's elsif. Thus

```
some condition
   some code
ELSE
   some condition
      some code
   ELSE
      some condition
         some code
      FIN
   FIN
FIN
```

can be equivalently written as

```
some condition
   some code
OTHERWISE
some condition
   some code
OTHERWISE
some condition
   some code
FIN
```

The following conditionals are available:

(A quick reminder: the value of a place or an object, is its state; however, variables (local or global) can be "pointers", i.e. have the refno of some entity is loaded into them. In the below, variables, which are in fact pointers are represented as "varname*".) value of a variable is either just that value or (if indirection is indicated

IFEQ {entname1|const1}, {entname2|const2}
: True if the value of the first argument is equal to the value of the second argument.

IFNE {entname1|const1}, {entname2|const2}
: True if the value of the first argument is not equal to the value of the second argument.

IFLT {entname1|const1}, {entname2|const2}
: True if the value of the first argument is less then the value of the second argument.

IFLE {entname1|const1}, {entname2|const2}

True if the value of the first argument is less then or equal to the value of the second argument.

IFGT {entname1|const1}, {entname2|const2}

True if the value of the first argument is greater than the value of the second argument.

IFGE {entname1|const1}, {entname2|const2}

True if the value of the first argument is greater than or equal to the value of the second argument.

IFINRANGE {entname|constant} {entname|constant} {entname|constant}

True if the value of the first argument is greater or equal to that of the second argument, but less or equal to the value of the third one.

CHANCE {entname|constant}

True with the probability of n%, where 'n' is the value of the argument.

IFHAVE {objname|varname}* [{state|flagname}]

True if the player is holding the specified object. If the second argument is supplied, the object also has to be in the specified state or have the specified flag set.

e.g. if THING is a variable,

```
LDA THING, BOTTLE
IFHAVE THING
```

will be true if and only if the bottle is in the player's inventory. Of course,

```
IFHAVE BOTTLE
```

will have the same effect in this particular example.

IFHERE {objname|varname*} [{state|flagname}]

True if the specified object is at the same location as the player. If the second argument is supplied, the object also has to be in the specified state or have the specified flag set.

IFNEAR {objname|varname}* [{state|flagname}]

True if the specified object is held by the player or is in the same location as the player. If the second argument is supplied, the object also has to be in the specified state or have the specified flag set.

IFFLAG {entname|varname*} flagname

True if the specified entity has the corresponding flag set. If a flagless entity is specified, the test returns FALSE.

IFAT {placename|varname*} [...]

True if the player is currently at the one of the specified locations.

IFLOC {objname|varname1}*, {placename|varname2*} [...]

True if the object specified by the first argument is at one of the the specified locations.

IFIS varname, {objname|placename|varname*} [...]

True if the named variable is a pointer to one of the specified objects or locations.

IFKEY word [word]

True if all specified words appear in the player's command.

IFANY word [word...]

True if any of the nominated words appear in the player's command.

QUERY {textname|varname*}

Displays the nominated text, which should be a yes/no question, and gets player response. Set to true if the answer is yes and false otherwise. **CAUTION** this directive is incompatible with library and single-turn modes (and thus incompatible with the HTML/JavaScript build, which uses the library mode). If such modes are to be supported, use the CONTEXT variable mechanism instead.

IFHTML
: True if the game is running in a mode in which its output is formatted as HTML.

IFCGI
: True if the game is running in a cloud via a CGI interface.

IFDOALL
: True if the game is processing a DOALL command loop.

IFTYPED
: True if the player actually typed the string given as an argument, as one of the command words.

    Thus for example

```
IFTYPED W
```

    will be true only if the player typed W rather than WEST, even though W is interpreted as WEST.

IFNEEDCMD
: True if there are no more pending simple commands to process, i.e. the player is about to be prompted for a new command. This is useful e.g. to avoid interrupting processing of a compound command with a spontaneous offer of help.

Logical operators

Now for some logical operators, to string the tests together, creating "compound" conditions. Note that tests are executed in the order in which they are encountered, with no precedence rules for operators and no bracketing. So, conceptually, A and B or C and D is understood as ((A and B) or C) and D. This may be unusual, but is, in fact surprisingly natural (or at least I found it so :-)).

AND
: "And" the test results so far with the following test.

OR
: "Or" the test results so far with the following test.

XOR
"Xor" the test results so far with the following test.

NOT
Invert immediately following test.

Logically enough, here are delimiters for conditional code.

ELSE
Execute the following code, up to the next FIN, if the latest (compound) condition returned false.

OTHERWISE
An equivalent of "else if" in C or elsif in Perl. Useful for avoiding deeply nested if-then-else constructs.

FIN
Delimits the code associated with the most recent (compound) condition. Can be used interchangeably with EOI.

Iteration

Next come the opcodes to do with iteration.

ITOBJ varname [{placename|varname*}] [objflag]
Execute the following code up to the matching FIN repeatedly, with the value of the nominated "loop variable" becoming a reference to objects satisfying the optional location and/of flag/state constraints (or all object, if no constraints specified) in the order of their declaration.

ITPLACE varname [{placename1|varname1*}, {placename2|varname2*}]
Execute the following code up to the matching FIN repeatedly, with the value of the nominated "loop variable" running through the specified range of locations (default all declared locations) in the order of their declaration.

ITERATE varname, {entname1|varname1*|const1}, {entname2|varname2*|const2}

Execute the following code up to the matching FIN repeatedly, with the value of the nominated "loop variable" running through all values from entname1 to entname2 inclusive. If either of the two range delimiting entnames is a variable, its value is used as the appropriate loop boundary (this may but need not be the reference number of some other entity). If either is a constant, the value of the constant is used. Otherwise the reference number of the nominated entity is used.

NEXT
CONTINUE

Skip the rest of the iteration block and proceed with the next iteration loop.

BREAK
LAST

Break out of the innermost iteration block.

DOALL [{placename|varname*}] [objflag]

DOALL starts off a do-all loop, in which the REPEAT cycle is repeated, but instead of querying the player for input, input is constructed out of the verb in ARG1 and the next object fitting the specified criteria. The loop is terminated either when no more objects fit the criteria or when the FLUSH directive is executed.

FLUSH

Abort the do-all loop if one executing and flush the command line buffer

---

Execution flow control

CALL {procname|placename|verbname|varname*}

Execute code associated with the named entity, which may consist of one or more separate, identically named chunks of code. These are executed in the order of their declaration, until either none left or one of RETURN, QUIT or QUIT-implying opcodes is executed in one of them.

PROCEED
Terminate the execution of the current procedure. If the procedure is one of a group of procedures of the same name, the next procedure in the sequence (in the order of declaration) is executed.

RETURN
Terminates the execution of the current procedure and of all procedures within the group of procedures of the same name

QUIT
Abort execution of the current procedure and restart the REPEAT loop at the first REPEAT procedure.

STOP
Terminate the whole program immediately.

Moving player and objects

APPORT {objname|varname*} {placename|varname2*}
Transport the indicated object to the indicated location.

GET {objname|varname*}
Transport the indicated object into the player's hands. Equivalent to "APPORT {objname|varname*}, INHAND".

DROP objname*
Transport the indicated object (presumed to be in player's hands (location INHAND) to the same location as the player). Equivalent to

```
IFAT {objname|varname*}, INHAND
  APPORT {objname|varname*} HERE
```

```
FIN
```

GOTO {placename|varname*}

Transport player to the indicated location.

MOVE [word [...]] {placename|varname*}

If no word list supplied or if any of the supplied words appear in the player's command, transport the player to the indicated location and restart the main loop (the REPEAT loop).Equivalent to (if a word list is present)

```
IFANY [word [...]]
   GOTO {placename|varname*}
   QUIT
FIN
```

SMOVE [word [...]] {placename|varname1*} {textname|varname2*}

If no word list supplied or if any of the supplied words appear in the player's command, transport the player to the indicated location and display the indicated text, then restart the main loop. If a word list is present, equivalent to

```
IFANY [word [...]]
   SAY {textname|varname1*}
   GOTO {placename|varname2*}
   QUIT
FIN
```

Generating and manipulating output

There are several directives for producing text output. All of them can be "qualified" for the purposes of "#" and "$" substitution and/or of the text switch mechanism. The VALUE qualifier is mandatory, but both SAY and QUIP can be used with a single argument. If the argument is an object or a place (possibly indirectly pointed to through a variable), the current state value of the object or place is used as the implicit qualifier. See the section on A-code texts for a full a explanation.

SAY {entname|varname*} [{entname|constant}]

Display text associated with the specified entity (text, object, place or any of these indirected through a variable). General rules for

displaying texts are covered in the section on A-code texts. However, objects and locations can have up to three separate texts applying to them. Please see major directives OBJECT and PLACE as well as the minor directive DESCRIBE.

RESAY {entname|varname*} [{entname|constant}]

Just like SAY, except that any text accumulated but not yet displayed is discarded first.

QUIP {entname|varname*} [{entname2|constant}]

Like SAY, but perform a QUIT having output the text

RESPOND word [word [...]] {entname|varname*} [{entname2|constant}]

Like QUIP, but null effect unless one of the listed words is present in the player's command.

APPEND {entname|varname*} [{entname|constant}]

Like SAY, but it first eliminates any trailing line feeds (if any) in the output accumulated so far and replaces them with a single blank. That is, it appends its text to the preceding paragraph.

DESCRIBE {placename|objname|varname*}

Outputs the longest available description of the indicated object or location.

VOCAB {objname|placename} [flagname] [textname]
VOCAB word {objname|placename} flagname [textname]
VOCAB [textname]

Used for displaying context-sensitive vocabulary. The first of the above three formats displays either primary name of the object or place in question, or the text specified as the last argument, unless the flagname argument is present and the flag in question is not set for that object or place. For example:

```
VOCAB CAGE, SEEN
VOCAB DWARF, SEEN, DWARF.VOC
```

The second format applies to words which are neither places nor objects, but their listing should be regulated by a flag setting of some object or place. For example:

```
VOCAB CHASM, SW.OF.CHASM, BEEN.HERE
```

The display, if any, is prefixed with a comma and a space, except for the first word displayed. The count of words displayed is reset to zero by a VOCAB directive with no arguments or with just the textname argument.

TIE textname [textname2...] [entname|textname]
Ties the value of the text (or texts) to that of the indicated entity or text. (See the A-code texts section for an explanation of text values.) The effect is that the value(s) of text(s) being tied are automatically kept in step with the value of the entity or text given as the last argument.

Arithmetical operations

SET entname {entname|constant}
Set the value of the entity given as the first argument, to the value of the constant or entity given as the second argument. Note that **no** indirection occurs with this opcode. If the first argument is an "indirector" variable, SETting it reverts it to an ordinary variable. Use the DEPOSIT opcode to SET with indirection.

ADD entname {entname|constant}
Increase the value of the entity supplied as the first argument, by the value of the constant or entity supplied as the second. Note, that **no** indirection takes place!

SUB entname {entname|constant}
Decrease the value of the entity supplied as the first argument, by the value of the constant or entity supplied as the second. Note, that **no** indirection takes place!

MULTIPLY entname {entname|constant}

Set the value of the nominated entity to its original value multiplied by the value of the second argument. Note, that **no** indirection takes place!

DIVIDE entname {entname|constant}

Set the value of the nominated entity to its original value divided by the value of the second argument. Note, that **no** indirection takes place!

INTERSECT entname {entname|constant}

Set the value of the nominated entity to the bit-wise "and" of its original value and the value of the second argument. Note that **no** indirection takes place!

NEGATE entname

Set the value of the nominated entity to its original value negated. Note, that **no** indirection takes place!

Randomisation

RANDOM entname {entname|constant}

Set the value of the entity indicated by the first argument to an integer chosen randomly from the interval between zero (inclusive) and the value of the entity or constant given as the second argument (exclusive). E.g. RANDOM CLOCK 10 will set the value of CLOCK at random to an integer number from 0 to 9. Note that **no** indirection takes place.

RANDSEL varname, entname1, entname2, [...]
RANDSELECT varname, entname1, entname2, [...]

Selects at random (i.e. with equal probability) one of the listed entities and makes its first argument into a pointer to that entity.

CHOOSE entnam {entnam2|constant2} {entnam3|constant3}

Set the value of the entity indicated by the first argument to a random integer from the interval indicated by the second and the third

arguments. If these latter arguments are both constants or variables, their values are used to determine the interval boundaries. However, for any entities other than variables, the reference numbers of the two entities are used instead. The randomisation is inclusive of the interval boundary values. E.g. CHOOSE VAR1 20 29 will set the value of VAR1 to a random integer between 20 and 29 inclusive, while CHOOSE VAR1 FIRST.QUIP, LAST.QUIP (where the two "quips" are not variables but text names) will load into VAR1 the reference number of an entity (text in this case) randomly chosen from between FIRST.QUIP and LAST.QUIP inclusive (there may well be some unnamed texts between these two).

RANDOMISE {objname|placename}, constant

Set state of the object or place to a random value between the base lower bound indicated by the constant (usually 0), and the highest numbered text switch component in any of the object's or location's description.

Refno manipulation

LDA varname, entname

Make the variable specified by the first argument "indirect" to to entity specified by the second.

EVAL varname varname

Set the value of the variable specified by the first argument to the value of the entity "indirected" through the second argument. Please note that no check is made to verify that the second argument "indirects" anything meaningful.

DEPOSIT varname {varname|constant}

Set the value of the entity "indirected" by the first argument to the value specified by the second. Note that no check is made whether the first argument truly indirects to some other entity.

LOCATE varname objname*

Make the indicated variable into an "indirector" for the location currently holding the indicated object.

Manipulating flags

FLAG entname flagname

Switch on the flag identified by "flagname" in the instance of the appropriate flag set belonging to the indicated entity.

UNFLAG entname flagname

Switch off the flag identified by "flagname" in the instance of the appropriate flag set belonging to the indicated entity.

Communicating with the machine environment

EXEC {varname|constant}, varname

Perform a special action, indicated by the value of the first argument, and return the result in the value of the variable given as the second argument. This opcode is to do awkward things, which are not allowed for in A-code opcode definitions or are far easier done in C than in A-code. All these special actions are performed in the adv00.c procedure special(). As supplied the following actions can be selected by the appropriate value of the first argument: the value

1. dump game to disc (obsolete – use SAVE FILE instead)
2. restore game from disc (obsolete – use RESTORE FILE)
3. delete saved game(obsolete – use DELETE FILE)
4. Obsolete – Adv550 legacy only (flush game cache)
5. Obsolete – Adv550 legacy only (get prime time flag)
6. save value of a variable (obsolete – use local variables instead)
7. restore value of a variable (obsolete – use local variables instead)
8. get number of minutes since restored game saved
9. set the value (pointer!) of ARG1

10. set the value (pointer!) of ARG2
11. pretend player said "X X" instead of "X"
12. check for end of player's (possibly compound) command (obsolete – use IFHAVECMD instead)
13. (spare)
14. retrieve a persistent data flag (should be part of IFFLAG?)
15. store a persistent data flag (should be part of FLAG?)
16. delete a persistent data flag (should be part of UNFLAG?)
17. save current location of all objects (should be part of SAVE?)
18. retrieve saved location of an object (should be part of RESTORE?)
19. toggle output text justification
20. set screen width (in fixed font characters)
21. set page margin (in fixed font characters)
22. set screen height (in lines)
23. save player's command (obsolete – use SAVE COMMAND instead)
24. restore player's command (obsolete – use RESTORE COMMAND instead)
25. (spare)
26. (spare)
27. {spare)
28. recover from failed restore
29. swap ARG1 and ARG2
30. (spare)
31. (spare)
32. check object being on the exception list
33. check existence of a memory save
34. list available saved games

Any number of other special action may be defined, but codes up to and including 100 are **reserved for future use** by the engine.

Get and manipulate player input.

INPUT [{textname|varname*}]

Input and parse a command, setting the mandatory variables ARG1 and ARG2 to the supplied verb and noun respectively. If only a single word is given and the mandatory flag PLS.CLARIFY of the mandatory

variable STATUS is set, the supplied word is combined with the last (incomplete) command.

Note that the player is queried for input only if we have run out of the last command line. There are three circumstances under which the player is not prompted:

1. He gave several commands separated by full stops or semicolons and we have not processed the last one yet.
2. The last processed command consisted of a verb followed by several nouns and we haven't yet finished applying the verb to all the nouns. This case does not preclude case (a) also applying!
3. We are in a doall loop, applying the verb of the last processed command to all objects satisfying the criteria of the DOALL opcode, which started the loop – and there is at least one more object to process. Again, this case does not preclude either of the two preceding cases applying simultaneously.

If the optional argument is supplied, the nominated text is displayed, before accepting player's input. Whether or not this happens, if the last printed text was a "fragment", it is pushed out as the prompt and padded with a space, if required. If the last message was not a fragment, the player is prompted by the standard linefeed and question mark prompt. Note that this allows you to have any prompt you like, instead of the standard one.

FAKEARG [{entname1|varname1*}] [{entname2|varname2*}]

If player's command refers to an entity referenced (possibly indirectly) by the first argument, pretend that the reference was to the entity referenced by the second argument. This directive does not, however, alter the words of the command.

FAKECOM [{entname1|varname1*}] [{entname2|varname2*}]

Like FAKEARG, except that the relevant word in the player's command is also modified, in case it is echoed back at the player.

VERBATIM {ARG1|ARG2}

Replaces the word string associated with ARG1 or ARG2 respectively, with whatever the player actually used, which got interpreted as whatever word string that is to be replaced.

UNDO

If undo is permitted, undo the number of commands specified by player command's second word, which may be (a) a number to undo a specific number of commands, (b) ALL to undo all commands since the last restore, or (c) UNDO to undo the immediately preceding UNDO. The UNDO_TRIM flag of the UNDO_STATUS variable is set or cleared appropriately.

REDO

Only accepted immediately after an UNDO command. REDO reverts the undoing of the specified number of commands, which may be given as (a) a number or (b) ALL which is equivalent to UNDO UNDO. The UNDO_TRIM flag of the UNDO_STATUS variable is set or cleared appropriately.

DEFAULT [{placename|varname*}] [objflag]

If no object has been specified in the player's last command (i.e. if the value of STATUS is 1), check whether there is an object at the nominated location (default HERE) and, optionally, has the nominated flag set. If no such object exists, this directive has no effect. If only one object satisfies the criteria, ARG2 is set as if the player had explicitly nominated that object. If more than one object fits the criteria, ARG2 is set to AMBIGWORD. In either case, the value of STATUS is increased from 1 to 2.

Saving and restoring (file, memory or command)

SAVE {FILE|MEMORY|COMMAND} varname

SAVE FILE saves the current state of the game in a file nominated by the command text string associated with ARG2. SAVE MEMORY creates or replaces an in -memory save image (this gets written off to

disk in ADVLIB and CGI modes). SAVE COMMAND saves player's parsed command, so that the game can ask a yes/no (or other) question, without relying on the QUERY directive, since this directive is not compatible with ADVLIB or CGI modes.

The variable specified by the <varname> argument returns zero on success. Non-zero return indicates failure.

RESTORE {FILE|MEMORY|COMMAND} varname

Restores file, memory image or player command saved by the SAVE directive.

The variable specified by the <varname> argument returns zero on success. Non-zero return indicates failure.

DELETE {FILE|MEMORY} varname

Deletes respectively a saved file nominated by the second word of the player's command, or the saved memory image (which also can be a file, in an ADVLIB or CGI modes).

The variable specified by the <varname> argument returns zero on success. Non-zero return indicates failure.

Debugging minor directives .

CHECKPOINT

SAYs its location (file name and line number) in the A-code source.

DUMPDATA

Dumps game data to STDERR or to the log file if one is being written to. If a second command word is given, it is taken to indicate the type of value-bearing entites to be shown. Possible values are "objects", "locations" or "places"), "variables" (or "vars") and "texts" – all of these being abbreviable to a single character. The command is handled by the kernel and hence types of dump need not be in the game's

vocabulary. Most likely use of this directive is in optionally included code – see the debugging section for debugging use of optional includes.

Obsolete and/or deprecated lexicals and directives

These directives are still supported for compatibility with old versions of A-code. Some are supported only for style 1 of A-code (i.e. only for Platt's original source).

Compound constants

A compound constant is a text string with no spaces, where names of simple constants or entitynames are joined by plus and/or minus signs.E.g. LAST.DEFLECTOR-FIRST.DEFLECTOR+2 evaluates as the value of LAST.DEFLECTOR minus the value of FIRST.DEFLECTOR plus 2. For entity names the value used in the calculation is the refno of that entity. In this example from Adv550, the two symbolic names happen to be text names.

SYNON {value|symbname}, synon [...]

Used to define symbolic name for constants or synonyms for already defined symbols.

BISET {entname|varname*}, flagname
BIS {entname|varname*}, flagname

Obsolete synonym of FLAG.

BITST {entname|varname*}, flagname
BIT {entname|varname*}, flagname

Obsolete synonym of IFFLAG.

BICLEAR {entname|varname*}, flagname
BIC {entname|varname*}, flagname

Obsolete synonym of UNFLAG.

LABEL procname

Synonymous with PROC.

NAME entname, qualifier
Synonym of SAY, except that the qualifier is mandatory.

DEFINE placename
Add the placename to player's vocabulary. The recommended way of doing this is by prefixing the placename with the + sign in its declaration

EOI
Closes an iteration loop. Use FIN instead!

EOF
Used in Platt's code as a short-hand for an arbitrary number of successive FINs.

DBNAME database-name
Defines the name of the data file.

TITLE title
Older synonym of DBNAME.

KEYWORD word [...]
If all indicated words (or their synonyms) appear in input, execute the following code, up to the next major directive; otherwise PROCEED.

HAVE {objname|varname*} [...]
If all indicated objects are held by player (at location INHAND), execute the following statements up to the next major directive; otherwise PROCEED.

HERE {objname|varname*} [...]
If all indicated objects are at the same location as the player (specified by the variable HERE), execute the following code, up to the next major directive; otherwise PROCEED. There was no corresponding opcode in the original A-code.

NEAR {objname|varname*} [...]

If all indicated objects are either held by the player (at location INHAND) or at the same location as the player (as specified by the variable HERE), execute the following code up to the next major directive; otherwise PROCEED.

AT {placename|varname*} [...]

If the player is at (any of) the indicated location(s), execute the following code, up to the next major directive; otherwise PROCEED. Don't worry about the name of this opcode clashing with the AT major directive – they exist in different name spaces, major directives being recognised by starting in column one. Note that the "place" may be an object!

ANYOF word [...]

If the player typed any of the indicated words in the command being processed, execute the following code, up to the next major directive; otherwise PROCEED. If successive lines have the ANYOF opcode, they are merged internally into a single ANYOF directive. If any of indicated words are in command do following; else PROCEED

VALUE entname* [{entname|constant}]

Like SAY, but replace '#' with value of the qualifier, rather than the qualifying entity name.

SVAR {varname|constant}, varname

Set value of the variable nominated in the second argument, according to the value of an environmental variable (or condition) indicated by the value of the first argument. Supported only for the sake of Platt's original code of Adv550, which uses SVAR 4 and SVAR 5 to regulate timing of game restores.

A-code history

The A-code language was created by Dave Platt in early 1980s for the purpose of writing his classic 550 expansion of the original Adventure. It

was subsequently expanded by myself in mid 1980s when merging Platt's Adventure 3 (now known as Adv550) with Luckett's and Pike's AdventureII (now known as Adv440) into Adv660 (well... into Adventure4, upgraded to Adventure4+, now known as Adv660).

I embarked on further expanding Adv660 into Adv770 in 1998, one of the purposes of the exercise being to explore further possibilities for improving A-code. The final result was the considerably improved Acode12. The history section describes the history of the language in greater detail.

A-code upward compatibility of saved games

A-code makes it fairly simple to preserve upward compatibility of saved games – a feature I found indispensable in alpha and beta testing, and generally useful after that.

You can just stick to following the five rules outlined below, and ignore ignore all accompanying explanations, but it may be useful to have some idea as to why those rules need to be followed. For that you need to have some grasp of A-code's internal concept of *refnos* (short for reference numbers).

Refnos were briefly explained in the main description of the A-code language, so just briefly...

A refno (or in full a reference number) is a number uniquely assigned by A-code to value-bearing declared entities – i.e. to objects, locations, variables and texts. The important point is that these refnos are allocated sequentially within each of these four categories in the order of entity declarations. Thus in a game source which mixes randomly object, place, variable and text declarations, objects will be numbered consecutively in their order of declaration, followed by places in their order of declaration etc.

It is by preserving associations of refnos to game entities *within* these categories that saved games are kept upwardly compatible. So let's consider what specifically would or would not break such compatibility. For simplicity, I'll stick with talking about objects, but exactly the same points apply within other categories that can have non-constant values associated with them - places, variables or texts.

If a previously declared object is removed in a later version of game's code, then all objects declared after the removed one will have their refnos reduced by one. If there are such objects, compatibility breaks. If there aren't (i.e. if the last declared object was removed, it does not break.

Similarly, if an object is added, then refnos of all objects declared later in the game's code will be increased by one. If there are such objects, compatibility breaks. If there aren't (i.e. if the added object is the last object declaration, it does not break. So...

Rule 1: do not add objects (places, variables or texts) except at the end of all object (place, variable or text) declarations.

What about removing objects. As should be obvious from the above, that's a bad idea. While it may seem that removing object(s) from the end of object declarations, the internal mechanics of the A-code kernel preclude this too. Thus...

Rule 2: The overall number of objects (places, variables or texts) may increase (as per rule 1), but decreasing it will break compatibility.

If you do find that an object is no longer required, don't remove it. Just leave it there, possibly giving it a different name, e.g. SPARE.OBJ.1 or something equally obvious. Better still, make its new name start with a dot: .SPARE.OBJ.1 -- this will stop the *acdc* translator complaining about an object being declared but not used. If later on you do want to re-use that spare slot for some other object, you can do so, but you cannot assume its value to be zero after a game is reloaded – that game may have been saved before you removed the original object.

As explained in the language description section, in addition to an integer value all objects, places and variables carry a bit-screen of flags. Flags have to be declared as symbolic names, but as far as the A-code kernel is concerned, such names are just synonyms for integer (non-negative) bit-screen offsets. Just as for refno association with entities, the association of these values with flag symbolic names must be preserved if saved games are upwardly compatible.

Therefore Rules 1 and 2, stated above, apply to flag declarations too (separately for object, location and variable flag sets), but there is an additional constraint, which can be stated as

Rule 3: do not change bit-screen byte-sizes!

Simple version: declare up front a few spare flags in each of the three categories – object, location and variable. To avoid *acdc* complaining about the extra flags not being used, give them names starting with a dot (e.g. .SPARE.OBJ.FLAG.1 etc) or wiith "spare.." (e.g. SPARE..O1).

Complex version: you may have some spare flags in each flag set, even if you did not declare any spare ones. This needs unpacking and to do so, I need to explain a bit about internal data structures involved.

While A-code only permits a single flags declaration for both of objects and places, multiple flag sets of various sizes can be declared for variables. It may seem puzzling that bit-screens carried by variables are not somehow associated with particular variables (except possibly via code comments). The simple explanation is that there is in fact just one bit-screen for variables too. All the separate declarations of it merely define different synonyms for the same bit offsets. Thus e.g.

```
FLAGS VARIABLE
      FLAG1
      FLAG2
FLAGS VARIABLE
      FLAG3
      FLAG4
      FLAG5
```

define one bit-screen of three bits, where FLAG1 and FLAG3 are both synonyms for 0, FLAG2 and FLAG4 are both synonyms for 1, and FLAG5 is a synonym for 2.

To make things more complicated, for historical reasons, location and object bit-screens automatically reserve the first three bits for kernel's use. Thus

```
FLAGS OBJECT
      OFLAG1
      OFLAG2
```

declares a bit-screen of **five** bits, with OFLAG1 as a symbolic name for 3 and OFLAG2 as a symbolic name for 4.

And as a final complication, bit-screens are actually allocated in 8-bit bytes. Hence in the above examples, the object bit-screen will have three spare

flags (offsets 5, 6 and 7), while the variable bit-screen will have 5 spare (offsets 3, 4, 5, 6 and 7).

We are now ready to calculate "bit-screen byte-sizes" that have to be preserved. If the longest variable bit-screen declared by game's code has N flags, the size of the bit-screen is the smallest number of 8-bit bytes containing at least N bits. For both place and object bit-screens it is the smallest number of bytes containing at least N + 3 bits.

Thus, for example, if you have 17 distinct (non-synonymous) object flags, the number of bits actually uses is 20, leaving you with four spare flags. Of course, there is nothing to stop you declaring explicitly some spare flags (with names prefixed with a . to keep *acdc* happy) for potential future use.

Rules 1 and 2 above have dealt with declared texts, but an additional, related consideration applies to in-line texts.

Rule 4: do not use text morphing features in in-line texts.

By this I mean texts which have an internal dynamic: incrementing, cycling or randomised – in the case of in-line texts signalled respectively by 'i:', 'c:' and 'r:' at the beginning of the text. The 'f:' prefix, signalling that the in-line text is a fragment, is benign.

The reason for this restriction is simple: by their very nature, the order of declaration of inline texts is what it is. While the *acdc* translator can group them all safely at the end of text refnos, it can do nothing about their ordering. Therefore adding an in-line text necessarily increments refnos of all succeeding in-line texts. If such texts have no internal dynamic, this is not a problem, since being nameless, they cannot be referenced from elsewhere in the code.

And that's really it. In practice, while working on and testing Adv770 I found that preserving upward compatibility between game versions was not hard. If the kernel ever changes in a way which would enforce upward compatibility break, this will be signalled by a change in its major version number.

A-code command parsing

While all command examples in this document are given in the upper case for ease of visual identification, A-code parser is case-insensitive: all commands are automatically forced into lower case.

- Where and when parsing takes place
  - The QUERY directive
  - The INPUT directive
- Core parser operation
  - Simple commands
  - Compound commands
  - Identifying vocabulary words
  - Treating blanks as command separators
  - Inverting verb/noun order
- Additional parsing features enabled by game code
  - Repeating commands
  - Non-vocabulary "nouns"
  - Handling IT
- Optional parsing features, available to game code
  - Orphan processing
  - Handling collective nouns
  - Using EXCEPT with collective nouns

Where and when parsing takes place

There is no separate A-code directive for player command parsing. Parsing takes place as an integral part of acquiring player input. There are two separate minor directives which do so: INPUT and QUERY. It is INPUT that does the heavy lifting. The much simpler QUERY directive is there merely as a convenience for asking yes/no questions. For reasons explained elsewhere it has to be avoided in A-code games, which are expected to be built in single-turn or library modes.

The QUERY directive

The QUERY directive is in fact one of the set of A-code conditionals. It takes as its single argument a text, which gets displayed as a prompt and expects yes or no in response. For example:

```
QUERY "Do you really mean it?"
   SAY "It shall be done, boss!"    # Player said yes
ELSE
   SAY "Oh, OK. You had me worried there for a moment."   # Player said no
FIN
```

Parsing is extremely primitive: any response starting with 'n' (or 'N') is taken to mean "no", and any response starting with 'y' (or 'Y') is taken to mean "yes". A null response is interpreted as "yes". Any other response is rejected with "*Eh? Do me a favour and answer yes or no! Which will it be?*" (this text is at present hard-wired into the kernel). If the player still responds with something that cannot be interpreted as yes or no, the parser says "*(OK, smartass... I'll assume you mean YES - so there!)*" (also hard-wired) and returns true.

The INPUT directive

The INPUT directive is the one that acquires and parses player commands. It takes as its optional argument a text (or a variable pointing to a text), which is to be used as a prompt, instead of the default question mark followed by a space. The rest of this document is devoted to parsing done by INPUT.

While the basic form of a player command has the simple "<verb> <noun>" format, the parser can handle compound commands, reducible to a series of simple command in the basic format. When a compound command is given, successive executions of the INPUT directive process the implied simple commands, one at a time.

The result of parsing a simple command is delivered in three variables: ARG1, ARG2 and STATUS. ARG1 and ARG2 contain respectively refnos of the verb and the noun of the simple command parsed. The STATUS

variable is set to the number of words in the command: one or two, or to the automatically defined constant BADSYNTAX a syntax error has been identified. For more detail see below, in the section on identifying vocabulary words.

Core parser operation

Conceptually, A-code command parsing has three separate components:

- Parsing that is performed by the INPUT directive regardless of the game code,
- Additional parsing features which are enabled by some particular entities being declared in the game code, and
- Optional parsing features made available by the kernel to A-code games.

This section deals with core parsing features, which are not influenced by the game code.

Simple commands

A command consists of one or more words (a.k.a. tokens), delimited by blanks (spaces) and/or commas (,) and/or dots (.) and/or semicolons (;). All leading and trailing blanks are ignored and multiple blanks are treated as a single blank.

A simple player command has the verb or verb/noun structure. E.g.

```
GET LAMP
```

Compound commands

Compound commands can be constructed by joining simple commands by command delimiters. The parser recognises three such delimits, which are completely synonymous: a semicolon (;), a dot (.) or ' then ' (note that

unlike the colon or the dot, the ' then ' delimiter must be surrounded by blanks, to make it into a separate token.

Compound command may also feature object iteration: a verb followed by a list of nouns, separated by commas (,) or ' and '. Again, the surrounding blanks are mandatory for the latter form.

Thus for example,

```
GET LAMP AND KEYS THEN READ POSTER
```

is equivalent to

```
GET LAMP, KEYS. READ POSTER
```

and is parsed as meaning

```
GET LAMP
GET KEYS
READ POSTER
```

In order to be maximally forgiving, the parser will also understand some incorrect but unambiguous variants on this syntax. E.g. in ' and then ' (equivalent to ',.') the iteration delimiter is ignored. So

```
GET LAMP AND KEYS AND THEN READ POSTER
```

will have the obviously intended effect.

The special English words AND and THEN are default equivalents of a comma and a semicolon respectively, but they can be replaced by others by a game's code. See the A-code internationalisation section.

Identifying command words

Individual command tokens are matched against the game's vocabulary. At game's discretion, the matching process may be

- restricted to exact matches, or
- permit minimal (automatically derived) abbreviations, or

- permit single typo correction – see the vocabulary description for more details.

See the A-code vocabulary description for details of command words matching against game's vocabulary.

If a command word is not found in the game's vocabulary and the ENTNAME variable is defined, the list of entity names is also searched, looking for an exact match.

If a simple command is parsed successfully, values of ARG1 and ARG2 are set to the refno values corresponding to the first and the second command token respectively. However, a further enhancement, if ARG1 value turns out to be in the range of object or location refnos, and ARG2 in the range of verb refnos, the two command words are swapped around. Thus, for example BIRD GET gets parsed as GET BIRD.

An additional parsing feature is present for games of style 11 or higher. If no match is found for the second command word, the kernel still does not give up. It is possible that the player was trying to reference something mentioned in an object or place description, or in some response recently given to an earlier command. Every time some text is output, all of its words longer than 3 characters and not ending in "ing", are stored in a separate, temporary vocabulary. This vocabulary, which is re-initialised whenever the player changes location, is scanned for an exact match (no abbreviations or typos). If a match is found, this is still treated as a matching failure, but of a different kind, so that if desired, it can be treated differently by A-code source.

If the parser fails to match a vocabulary word, the corresponding ARG1 or ARG2 variable is set to one of special pre-defined values:

- BADWORD – no match of any kind.
- AMBIGWORD – abbreviation matching gives a non-unique result – more than one vocabulary word could be meant.
- AMBIGTYPO – single typo matching gives a non-unique result.
- SCENEWORD – the match is not against the vocabulary by against the list of words used by the guide since the last change of location.

- BADSYNTAX – any other parsing failure. (Note that in this case the STATUS variable is also set to BADSYNTAX.)

Finally if a command consists of just a single word (the STATUS variable is set to 1), the ARG2 variable is set to the predefined constant NOWORD.

Treating blanks as list or command separators

The verb SAY is treated by the parser as a special case in that spaces can be used instead of commas as in SAY FEE FIE FOE, which is treated as SAY FEE, FIE, FOE (which is equivalent to SAY FEE; SAY FIE; SAY FOE).

Inverting verb/noun order

By default, in two word commands, the first command token is considered to be the verb and the second one as the noun, e.g. TAKE BOTTLE. However, if the first token is not likely to be a verb (being e.g. a place or an object) and the the second is identifiably a verb, the parser will automatically swap them around, making BOTTLE TAKE also a legitimate command.

Additional parsing features enabled by game code

Some functionality of the A-code kernel is only present if game source defines particular entities.

Repeating commands

If game source declares the word AGAIN (possibly with some synonyms), commands can be repeated by using AGAIN as a verb. If used within a compound command it will repeat the last sub-command delimited by THEN (or a dot or a semicolon). If used on its own, it will repeat the whole

of the player's last input, which may be a compound command (and may itself contain AGAIN on order to repeat its sub-command).

The special English word AGAIN is merely the default "repeater" word.

So, for example:

```
GET BUCKET.DRINK BUCKET. AGAIN
```

is equivalent to

```
GET BUCKET
DRINK BUCKET
DRINK BUCKET
```

whereas

```
THROW AXE THEN GET AXE
AGAIN
```

is equivalent to

```
THROW AXE THEN GET AXE
THROW AXE THEN GET AXE
```

Non-vocabulary "nouns"

Sometimes it is not desirable to match player input against the games vocabulary at all. Saving and restoring games is an obvious example of this – players cannot be restricted to game's vocabulary for naming saved games. One could, of course, insist that SAVE and RESTORE do not take a save name as the second word of the command, but invoke instead a separate input routine, which takes the desired save name without any reference to the vocabulary. However, the same applies to any command which takes a numerical argument, e.g. specifying screen width or height in the console mode.

Rather than having special code for handling such (and similar) exceptions, A-code's approach is to tag relevant verbs as "special". In the absence of flag settings for vocabulary words (possibly to be rectified in the future), the solution is to groups declaration of such verbs between declarations of

two pseudo-verbs: first.special and last.special. This works because (a) A-code allocates refnos (in a given entity category) in the order of declarations and (b) if prefixed with '-' these pseudo-verbs are themselves allocated refno, but are not added to the game's vocabulary. Here's an example based on Adv770:

```
# The next block are specials, not requiring validation of ARG2.
#
verb -first.special        # Mark the first one
verb again, repeat, =r
verb save, suspend, pause
verb restore, load
verb rest, wait            # In case players type REST MYGAME
verb !length, =!line, =!width
verb !scroll, =!screen, =!depth
verb !margin, =!offset
verb restart, initialise
verb why
verb please
verb -last.special         # Mark end of special verbs
#
# End of verbs not requiring validation of ARG2.
```

The kernel is aware of the special significance of first.special and last.special and will automatically suppress validation of the second command word when parsing a command with any verbs defined as special in this manner. However, a similar mechanism can be profitably used by A-code source. Here's another example from Adv770:

```
verb -first.direction
verb north, =n
verb northeast, =ne
verb east, =e
verb southeast, =se
verb south, =s
verb southwest, =sw
verb west, =w
verb northwest, =nw
verb -last.compass.point
verb up, =u, upward, ascend
verb down, =d, downwards, descend
verb -last.direction
```

This makes it easy to check for a command word being a direction

```
IFINRANGE ARG2, FIRST.DIRECTION, LAST.DIRECTION
```

or a compass point:

```
IFINRANGE ARG2, FIRST.DIRECTION, LAST.COMPASS.POINT
```

Handling IT

Handling the indexical noun IT is fairly straightforward in A-code. One declares a dummy object of that name (with whatever synonyms may be deemed appropriate) and sets its value to be a pointer to an appropriate object. The value of IT should be set to a pointer to an object in the following situations:

- A player command explicitly names an object.
- Player's inventory gets listed and the list consists of a single object.
- Objects in the current location get listed and the list consists of a single object.

OTOH the value of IT should be probably cleared (set to zero) if either of the two kind of object list contain more than one item.

If a player's command contains the word IT and the IT object has a non-zero value FAKECOM can be used to set ARG1 or ARG2 (as appropriate) to pretend that the command explicitly named the object pointed at by IT.

None of the above involves any special kernel functionality, so why is the matter even mentioned in this description of command parsing by the kernel? The reason is that there is one other situation which requires the value of IT to be modified, and that situation is handled by the kernel.

If IT is declared as an object and the game source successfully uses the DEFAULT directive to select a default object, then IT is automatically set by the kernel to point at the object defaulted to.

Optional parsing features available to game code

Other features of the A-code kernel, which are also dependent on some entities being or not being defined in game source, do not take effect automatically, but have to be explicitly triggered by game code as and when appropriate.

Orphan processing

If a player issues a single word command, which can be assumed to imply a a target object on an action (e.g. PUSH on its own, or KEYS ditto), the game can do better than just complain that not enough information has been given. If the A-code source declares PLS.CLARIFY as a flag, setting this flag on the automatic variable STATUS affect the way the next command gets parsed.

If the next command also consists of a single word, the "orphan processing" mechanism comes into play. It combines this command with the preceding one-word one, so that the player does not have to repeat the previously typed word. On the other hand, if the next command consists of two words, it is parsed in the usual manner and the clarification request is ignored. The PLS.CLARIFY flag gets automatically unset in either case, so that it does not have to unset explicitly by the game's code.

Here's an example of orphan processing in action:

```
? get
What do you want me to get?
? rod
You get the rod.
? rod
What do you want me to do with the rod?
? drop
You drop the rod.
?
```

The underlying code dealing with generic GET requests could look like this:

```
action get
   ifeq status, 1
      default portable            # Find default object flagged as portable
      ifeq arg2, ambigword        # If more than one possible target
         flag status, pls.clarify # Activate orphan processing
         quip "What do you want me to {arg1}?"
      fin
......
```

The code for handling the drop command in the above example would have to come into play, once the game fails to find an action associated with the word ROD.

```
   call arg1                   # If arg1 has an associated action, execute it
   ifflag arg1, object         # We fell through, so presumably no such action
      flag status, pls.clarify # Activate orphan processing
      quip "What do you want me to do with the {arg1}?"
   fin
```

Handling collective nouns

The A-code language makes it easy for a game to permit use of collective nouns such as, for example, ALL or TREASURE. This is done by using the A-code directive DOALL before handling the verb to be applied to a collective noun. Here, for example is a very simple code to handle GET ALL:

```
action get
#
# Check for the command being GET ALL.
#
  ifkey all
    doall here, portable  # Sets up the "do all" loop
  fin
#
# The rest is ordinary handling of GET
#
  ifeq status, 2          # Do we know what to get?
    and
  ifhere arg2             # Is the the object in the same place as the player?
    and
  ifflag arg2, portable   # Is the object portable?
    apport arg2, inhand   # If so, relocate the object to player's possessions
    quip "You {arg1} the {arg2}  # Report the action.
  fin
fin
```

The DOALL directive in the above example sets up the do all loop of command processing, with the effect of setting ARG2 to the first object matching the specified criteria (being both co-located with the player and flagged as portable), and sets the value of STATUS to 2. The effect of this is that the rest of ACTION GET code results in that object being picked up and the REPEAT code loop restarted (due to the use of QUIP instead of SAY when reporting the action).

Because of the do loop being active, instead of prompting the player for the next command, the kernel constructs that command by taking the same verb (GET in our case) and combining it with the *next* matching object, if any. That command is again processed in the normal way. This continues until

such time as there are no matching objects left, at which point the do all loop is terminated and the player prompted for a command.

This is obviously a very simplistic implementation and various checks may be necessary, such as e.g. checking that the player has a spare carrying capacity for the next object to be picked up. Such checks cab be performed as appropriate using the IFDOALL directive, which executes its associated block of code if the do all loop is active. If needs be, the loop can be aborted by the FLUSH directive.

TREASURE could be handled in a similar manner, assuming that in addition to being flagged as being portable (via the game-defined flag PORTABLE in the above example), it is also flagged as valuable by some other game defined flag.

Using EXCEPT with collective nouns

As a further enhancement of handling collective nouns, if EXCEPT is declared as a vocabulary word, the parser accepts an extension of the simple verb/noun command structure. Where a collective noun (e.g. ALL) is allowable as a command noun, it can be followed by EXCEPT (or a synonym thereof) and a list of entities to be exempt from the requested action. For example:

```
DROP ALL EXCEPT LAMP AND KEYS THEN READ POSTER
```

will be understood in a natural manner.

This expanded syntax has to, of course, cater for the possibility of unrecognised words being given by the player in the list of exceptions. To do so, this syntax enhancement brings into existence the automatic variable ARG3. If a word in the exception list is not recognised, the do-all loop is aborted and ARG3 is set to an appropriate error code, and its associated word is the unrecognised word of the player's command. Here is an example of this kind of error handling in Adv770:

```
   ifkey all
      call shadow.shutup
```

```
ifeq arg3, badsyntax
   quip no.except, arg3
fin
ifeq arg3, ambigword
   quip tell.me.more, arg3
fin
ifeq arg3, badword
   flush
   quip nocomprende.object, arg3
fin
ifeq arg3, ambigtypo
   quip is.it.a.typo? arg3
fin
....
```

A-code internationalisation

As of version 12.83, A-code is sufficiently UTF8 compliant to handle games with messages, vocabulary and entity names in languages other than English, including ones which use non-ASCII characters, provided (a) words are separated by ASCII blanks (octal 32) and (b) are parsed left to right. All you need is a UTF8-compliant text editor. While this degree of UTF8-compliance is only possible since A-code version 12.83, it is available in games using A-code styles from 10 upwards.

UTF8 compliance means that you can have texts and player vocabulary, including object/place names in any character set that can be represented in UTF8 encoding of Unicode. To activate this feature of the current A-code engine, add the UTF8 major directive to the game source header.

Of course, full game internationalisation has other challenges, which are nothing to do with character encoding.

- Parsing of complex commands
  Some words, such a AND, THEN, AGAIN... have special meaning for the kernel. Different words will apply in different languages.
- Modifying kernel messages
  While A-code leaves most of communication with the player to the game's code, there are a few occasions when the kernel responds directy -- e.g. about being unable to save a game. These kernel messages also need changing for languages other than English.
- Echoing player's words grammatically
  When the game incorporates player command words into its responses, in English just echoing player's words is sufficient. Other languages typically need to be able to modify echoed words accoding to their grammar rules.

Parsing complex commands

A-code kernel has to be able to identify words being used to structure complex command: AND being used instead of a comma, and THEN being used instead of a semicolon. Furthermore, if the verb AGAIN is defined by the game, it is intercepted by the kernel and taken to men a request to repeat the previous command (see the section on automatic entities and flags in the A-code language documentation). There are a few other special words that have to be known to the kernel. The current complete list is as follows:

- **AND** – parsed by the kernel as equivalent to a comma (see the explanation of compound player commands).
- **THEN** – treated as equivalent to a semicolon (see the explanation of compound player commands).
- **AGAIN** – its presence in player vocabulary activates the repeater feature of command parsing (see the relevant part of the section on command parsing).
- **ALL** – its presence in player vocabulary activates the availability of the A-code minor directive DOALL (see the the relevant part of the section on command parsing.)
- **EXCEPT** – used to signal exceptions to an DOALL command processing (see the the relevant part of the section on command parsing.)
- **IT** – maintained by the kernel as a pointer to the last referenced objects (see the the relevant part of the section on command parsing.)
- **YES** and **NO** – have to be understood by the kernel for processing of the QUERY directive, which expects a yes/no answer.
- **UNDO** – its presence in player vocabulary activates the system for undoing/re-doing player commands (see the the explanation of the undo mechanism.) Note that it is not necessary to translate the REDO command.
- **SAY** – use in kernel is experimental, will be fully described in a later A-code version.

The obvious solution in writing a game in a language other than English is to declare the appropriate synonyms for such special words. (AND and THEN are defined automatically, but can be also defined explicitly as vocabulary words.) E.g. in Czech the equivalent of AND is A, and of THEN is PAK (or POTOM). Thus

```
WORD AND, A
WORD THEN, PAK, POTOM
```

enables these words to be used in place of AND and THEN.

There is, of course, an obvious snag to this simple solution. It leaves the English version of such special words in the player vocabulary, which may be very confusing to players (e.g. because of the typo correction mechanism). Or worse, those words might mean something else in some other language. To avoid such problems, one can "bar" English versions by prefixing them with the vertical bar ASCII character (|).

```
WORD |AND, A
WORD |THEN, PAK, POTOM
```

This makes the synonyms of a barred special word to be have the same effect, without adding the barred word itself to the list of defined terms. This makes it possible for that word to be defined in its own right and having another meaning altogether. As an example, the following is perfectly lega:

```
WORD |THEN, PAK, POTOM
WORD THEN
```

In this case words PAK and POTOM do have the special meaning of separating simple command within a complex command, but THEN itself has no special meaning at all. Any code references to THEN would then be referring to that last definition. This avoids any potential case of a clash between a reserved English vocabulary word and some word in another language.

If no synonyms are given, the excluded word can still be used by game code, if required.

Modifying kernel messages

There are some messages that the kernel does produce directly, instead of signalling its response to the game and letting the game to produce an appropriate message to the player. An obvious example is a query about

savign a game under the same name as an already existing save: "*There's already a game dumped under that name.*" follwed by "*Do you really mean to overwrite it?* ". Clearly some way is required for modifying such messages if A-code is to be used to write a non-English game.

From version 12.50 onwars, the kernel does provide an appropriate mechanism. Before displaying such a hardwired massage, the kernel first checks whether the game code has supplied an equivalent. If the equivalent exists, that is what is displayed. Otherwise the hard-wired default English message is shown

For example, if the game source defines a text named .K.REALLY.OVERWRITE, that text will be displayed instead of the default "Do you really mean to overwrite it? ". Each hard-wired message can be overriden by an appropriate .K.<some_name> text. You can find the full current definitions of such names and their default English texts in the appendix to this section.

You can also find that list as *i18n_en.acd* file in the *extras* sub-directory of the sources drectory of *acdc*. This file gives proper A-code definitions of .K. texts and can be (but dos not need to be) used as an include file in a game source. Some of these texts are only relevant for particular builds of the game, e.g. QT, or Windows console ones. Others will depend on use of a particular A-code feature, such as using the QUERY minor directive or replacing it with the CONTEXT mechanism).

The file is provided merely as a handy template for internationalising (or just customising!) kernel messages. To modify just some of those messages you only need to provide the definition of those specific message in a game's A-code source. It is only when writing a game in a language other than English that you should translate the whole of that template file.

As an illustration, alongside the *i18n_en.acd* the A-code tarball also contains a Russian version, kindly provided by Алексей Галкин as a part of his translation into Russian of my A-code implementation of Roger Frith's demo adventure *The Cloak of Darkness*.

Echoing player words grammatically

Echoing player words in Engish is easy enough. For example, if the player says TAKE CLOAK, responding with "You take the cloak" is fine, but in other langauges, player's CLOAK may have to be transformed from the nominative case to the genitive one. End even in English one may wish to reply with a plural (e.g. "There are no cloaks to be seen here") or transform an irregular verb into the past tense.

The current version of A-code does not as yet offer a way to do this, but I have a cunning plan :-) for a simple extension of the language which would deal with the easier aspect of this problem. Watch this space!

In the meantime you'll have to stucture game's responses to avoid the need for grammatical transformations.

Appendix: .K.* kernel text messages

```
text .K.DATA.DUMPED
  Game data dumped into the log file.
text .K.EXIT.ENTER
   (To exit, press ENTER)
text .K.IN.PROGRESS
  Restoring game in progress...
text .K.LOST.SESSION
  Oops! We seem to have lost your current game session! Sorry about that!
text .K.AS.YOU.WISH
  As you wish...
fragment .K.REALLY.OVERWRITE
  Do you really mean to overwrite it?_
text .K.ALREADY.HAVE.NAME
  There's already a game dumped under that name.
text .K.CHANGED.MIND
  Changed your mind, eh? Fine by me...
fragment .K.NAME.TO.RESTORE
  Name of saved game to restore:_
text .K.GAME.LIST
  You have the following saved games:
text .K.NO.SAVED.GAMES
  You have no saved games to restore.
fragment .K.NAME.TO.SAVE
  Name to save game under:_
fragment .K.YES.NO
  Eh? Do me a favour and answer yes or no! Which will it be?_
text .K.ASSUME.YES
   (OK, smartass... I'll assume you mean YES - so there!)
text .K.OK
  OK.
```

```
fragment .K.MORE
  [More?]
fragment .K.VERSION
  version
fragment .K.SUBMIT
  Submit
text .K.BROWSER.WELCOME
  Welcome to the A-code browser interface
text .K.BROWSER.INTRO
  This interface can be used entirely from the keyboard, as long as the
  command input field remains in focus. Commands can be submitted by pressing
  the RETURN key and arrow keys can be used to recall and edit previous
  commands.

  The appearance of the game can be customized by modifying the
  A-code configuration file acode.conf - this can be found in the
  A-code directory, called either .acode or acode, within your
  home directory. There you can also find individual game directories called by
  the game name, which is where saved games and log files are stored by
  default.

  The configuration file has ample comments to guide you, should
  you decide to make any changes. If by any chance this interface does not work
  for you, you can change the browser specification in that file to NONE and
  then run the game in the console mode from a terminal window.
fragment .K.BROWSER.PLAY
  Play
text .K.BROWSER.CODA
  (In case you are wondering why it is not possible to skip this intro page
  - it's because it is in fact a cunning workaround for a browser
  mis-feature, no names named. By clicking on the 'Play' button you will ensure
  that the browser behaves itself and gives keyboard focus to where the
  interface requests it to be.)
text .K.HTTP.INTRO
  The game can be played entirely from the keyboard, as long as the command
  input field remains in focus. Commands can be submitted by pressing the
  ENTER key (a.k.a. RETURN). Arrow keys allow recalling/editing previous
  commands. If you wish to stop playing and to continue at some other time,
  it is not necessary to use the save command - just open this page again
  and you will be offered the option of continuing the current game, if there
  is one. The game runs locally on your machine - there is no network access.
  All data storage relies on your browser's sand-boxed data store.
fragment .K.HTTP.CODA
  This JavaScript version of the game is made possible by the magic of
fragment .K.YOU.CAN
  You can
fragment .K.OR.LOAD
  or load a saved game
fragment .K.CONTINUE.GAME
  Continue your game
fragment .K.START.NEW.GAME
  Start a new game
fragment .K.OR
  or
text .K.CLOSE.BUTTON
  Close this tab/window
text .K.BROWSER.PARANOID
  (This will only work is your browser
 is not being paranoid about security.)
text .K.WRONG.VERSION
  Wrong game version!
```

```
fragment .K.PLAY.AGAIN
  Play again
```

A-code undo/redo facility

If the game defines UNDO as a verb, then the undo/redo A-code mechanism gets activated for that game. The REDO verb is then defined automatically, though it can be also defined by the game explicitly as a verb. In such circumstances, defining REDO as anything other than a verb is treated as an error. REDO can be used immediately after an UNDO command, to undo some or all of the UNDO.

The general form of UNDO and REDO commands is as follows:

verb UNDO [{<number>|ALL|UNDO}]
verb REDO [{<number>|ALL}]

The default number of turns to undo or redo is one, though a specific number can be specified. ALL means just that – undo all collected turn history or undo all of the preceding UNDO. "UNDO UNDO" is equivalent to REDO ALL.

Information on the effects of an UNDO/REDO command is returned by an automatic variable UNDO.STATUS, which may but need not be explicitly defined by the game's code. The variable is considered to have four automatically declared flags:

UNDO.INFO
: This flag is defined but not used by the kernel. It may be used by the game's code e.g. to enable or disable use of UNDO and REDO commands.

UNDO.TRIM
: Set if requested number of turns has been trimmed. The value of UNDO.STATUS is set to the actual number of turns undone or redone.

UNDO.INV
: Set if inventory contents changed, so that the game can note the fact. There is no equivalent flag for change of location, because that can be

checked by the game's code by preserving the HERE value in a local variable and then it with the value of HERE after the UNDO/REDO.

UNDO.BAD

Set if the UNDO/REDO is given an illegal argument.

Undo history is saved and restored as a part of game saving and restoring. However, should the game's number of objects, locations or global variables change, the undo history of games saved by an earlier version will be quietly ignored.

Here is a simple example of using UNDO:

```
action undo
  local uhere
  set uhere, here       # Preserve value of HERE in a local variable
  undo arg2             # Do the UNDO!
  ifgt undo.status, 0   # Something got undone
    ifflag undo.status, undo.trim
      say "Cannot undo that many turns!"
    fin
    say "Turns undone: $n.", undo.status
    ifne here, uhere
      say "You have relocated!"
    fin
    ifflag undo.status, undo.inv
      say "Your possessions have changed!"
    fin
    quit
  else                  # Nothing undone
    quip "Nothing happens."
  fin
```

A-code vocabulary

A-code vocabulary structure and handling have some unusual features, motivated by my desire to make IF games more player-friendly and less 'mechanical' in their responses to player commands. This document deals specifically with vocabulary handling, rather than with the more general topic of player command parsing. See the parser section for parsing details.

There are four specific areas in which A-code vocabulary handling is unusual. These are:

- Automatic abbreviations
- Approximate matching (a.k.a. typo correction)
- Synonyms and sub-synonyms (a.k.a. the 3D vocabulary structure)
- Vocabulary listing

Automatic abbreviations

By default, all vocabulary words are automatically abbreviated to the shortest unambiguous length. For example, if the vocabulary contains **KICK** and no other word beginning with **K**, then **KICK** is automatically abbreviable as any of **K**, **KI** and **KIC**. If, however, the vocabulary also contains **KILL** (and no other word beginning with **KI**), then **KICK** is only abbreviable to **KIC**, and **KILL** to **KIL**.

There are, of course, command words which must not be abbreviated: this is particularly true for any magic words. E.g. we do not want **X** to mean **XYZZY**! This is catered for in vocabulary declarations by prefixing the non-abbreviable word with an exclamation mark:

verb !xyzzy

This does not, however, eliminate all unwanted abbreviations. For example, a player may wish to save a game under some arbitrary name, which just happens to be a legitimate abbreviation of some vocabulary word. Clearly,

the intended name should not be expanded to the vocabulary word in question.

This problem is solved by permitting some vocabulary verbs to be marked as "special" in that other words in the command are taken verbatim, as entered by the player (except for the case, which is always normalised to lower case). For historical reasons, instead of such special verbs being flagged individually, at present (A-code version 12) they are grouped in one block of verb declarations:

verb -first.special
[standard declarations of any special verbs]
verb -last.special

The '-' sign preceding "first.special" and "last.special" indicate that these are pseudo-verbs, not to be added to the game's vocabulary. This is an extension of the same convention in declaring object names.

To cater for any other instances in which abbreviation expansion is not wanted, setting the mandatory STATUS variable to the built-in value of NO.MATCH suppresses expansion for the duration of one command cycle.

As a further wrinkle, for object names, abbreviation matching is performed only if the object in question is either carried by the player or is in the same location as the player. This prevents game secrets being inadvertently revealed by being too generous in interpreting player's command.

See the section on command parsing for an explanation of A-code's response to ambiguous abbreviations.

Approximate matching

My typing is terrible. From seeing many player logs I know that I am not alone in this. That's why by default A-code enables automatic approximate matching of words entered by the player against words in the game's vocabulary. This feature is activated by the game declaring a text named TYPO, the assumed purpose of which is to report typo matching. This is

necessary because otherwise typo correction can look distinctly baffling to players.

If a single typo is sufficient to transform an unknown word type by the player into a unique, non-abbreviated match in the vocabulary, that match is accepted as the player's command word. Here by a typo I mean a single character dropped, a single character interpolated, a single character substituted by another character or two adjacent characters swapped around. Thus, for example GET BOTTEL will be understood as GET BOTTLE.

Note, however, that approximate matching is attempted instead of, rather than in addition to unique abbreviation matching, if no unique abbreviation is found. Thus if BOTT is a unique abbreviation of BOTTLE, then GET BOTT will be understood, but GET BORT will not. That's because BORT differs from BOTTLE by three typos rather than one: mistyped R instead of T and two missing letters L and E at the end.

Clearly enough, the same problems arise here as with abbreviation matching. These are dealt with by the same mechanisms. No approximate matching in the following circumstances:

- word declared as non-abbreviable
- word accompanies a verb declared as "special"
- word refers to an object not yet seen by the player
- the STATUS variable is set to NO.MATCH or NO.AMATCH (duration one command cycle)

See the section on command parsing for a detailed explanation of A-code's responses to approximately matched commands.

Synonyms and sub-synonyms

Traditionally, IF game vocabulary is a 2D structure, consisting of list of lists of synonyms. Thus, for example, using A-code notation

verb get, take
verb drop, release

declares two vocabulary terms, both of which are denoted by two synonymous words. From version 10, A-code adds a third dimension: any individual word in the traditional structure may be associated with a list of sub-synonyms. For example, in Adv770 the full definitions of get and drop are as follows:

verb get, =g, =reach, carry, take, =t, pickup, keep, hold, catch, grab, =grip, clutch, steal, capture, tote, scoop
verb drop, =dr, =discard, =fall, =abandon, free, =release, =let

The '=' sign preceding a word indicates that word to be a sub-synonym of the last preceding word not prefixed with '='. So G and REACH are sub-synonyms of GET, but CARRY is not.

This additional vocabulary structure is motivated by my preference for responding to player commands rather more explicitly than is traditionally the case. Thus instead of the standard and rather mechanical "Taken." in response to GET CAGE, I like the game to respond with "You get the cage.". But there is a very obvious snag to this: G CAG should not result in "You g the cag."

While the expansion of "cag" to "cage" is handled by the abbreviation processing feature of the parser, "g" is declared explicitly as a vocabulary word, so does not get expanded as an abbreviation. If there were no other verbs beginning with g, it would not be necessary to include "g" as a verb synonym – the abbreviation processor would do the expanding, but that's not a realistic solution. Hence the declaration of "g" as a sub-synonym of "get", which instructs the A-code engine to echo the latter if the command verb is to be echoed.

Vocabulary listing

To my mind, one of the most irritating features of many IF games is the need to guess what words may or may not be present in the game's vocabulary. A-code offers an intelligent way of making the vocabulary listing available to players, without revealing things players should not as

yet know about. This is achieved by the minor directive VOCAB, which is a kind-of fancy wrapper around the SAY directive. The full syntax of the directive is as follows:

VOCAB {<OBJECT>|<PLACE>|<WORD>} [, {<PLACE>}] [, {<FLAG>}] [, {<TEXT>}]

The first argument is the vocabulary word to be conditionally included in a vocabulary listing. The optional second and the third arguments specify conditions under which the word is to be included. The fourth argument, also optional, specifies the text to be shown instead of the word given by the 1st argument. Each vocabulary word shown to the player is prefixed with a comma followed by a space. A special form of the directive

VOCAB [<TEXT>]

is used to signal the start of a sequence of VOCAB directives. It can optionally display a text, but its main purpose is to suppress the comma-and-space lead of the following vocabulary list.

As an example, here are some VOCAB commands from Adv770

vocab voc.lead
: A special form, signalling the beginning of a vocabulary listing in order to suppress the leading comma before the first word to be shown; in this example a text name is given as an argument, so the **voc.lead** message will be also displayed.

vocab axe
: The word **axe** is shown unconditionally.

vocab alarm, seen
: The word **alarm** is shown if the *seen* bit of the *alarm* object is set.

vocab bones, bones.room, been.here
: The word **bones** is shown if the player has been at the *bones.room* location.

vocab eggs, seen, eggs.voc

The text *eggs.voc* (in this case "eggs (nest)") is shown if the object *eggs* has been seen.

A-code texts explained

A-code texts are de facto objects with their own methods, making them into a far more powerful and flexible game component than in any other IF system. Yet they can be treated also as simple text strings – the complexity of "text morphing" is there to be used, but it does not force itself onto game writers.

This section explains A-code texts and their handling as of A-code version 12.91.

General notes on style

- The use of upper/lower case in the examples is purely my programming convention; except within text definitions, A-code is case insensitive.
- The use of dots to separate words in A-code entity names is also conventional; any other convention could be used, e.g. using dashes or underscores instead of dots, or (not recommended!) uppercasing the first letter of each word.
- The use of commas as A-code statement parameter separators is purely optional, for increased readability; A-code counts commas to be white spaces.
- All examples use the up-to-date A-code notation (A-code 12), which differs in some respects from that in the original version of the language by Dave Platt.

A-code text basics

This part of the section deals with the basics of A-code text declaration and use.

A-code text declarations

As of version 12, in-line texts are also permitted (see the in-line-text section), but traditionally A-code defines all its texts (other than place and objects descriptions) as separate named entities. The basic A-code text declaration has the form

```
TEXT <text_name>
   <some_lines_of_text>
```

Like all major A-code directives, the keyword TEXT must be at the very beginning of a line. The text lines following it must all have at least one leading blank. The declaration is terminated by a line with a non-blank first character (i.e. another major directive or a comment), or by end-of-file.

While generally line breaks are ignored in the text definition (see the next section for more details), any texts declared using the TEXT directive are deemed to have a trailing end-of-line. A text fragment, without the trailing EOL can be declared using the FRAGMENT variant of a text declaration:

```
FRAGMENT <text_name>
   <some_lines_of_text>
```

Here is a simple, purely artificial example:

```
FRAGMENT LINE.START
   This line is sp
TEXT LINE.END
   lit into two parts.
```

When displayed by the A-code primitive SAY

```
say line.start
say line.end
```

The result is a single line "This line is split into two parts." followed by a line break.

Both TEXT and FRAGMENT directives can be used to declare "anonymous" texts by omitting the text name. This practice goes back to Dave Platt's original version and is now deprecated -- text switches can be used instead. See Appendix C on handling anonymous texts.

Basics of text definitions

A-code text definitions (the lines of text following the TEXT or FRAGMENT declaration line), are processed as follows:

- Any leading or trailing spaces on an individual line are removed.
- Simple line breaks (i.e. single end-of-line characters) are replaced with a space, *after* the removal of trailing characters. (But see the below discussion of text switches for an exception to this rule.)
- Successive line breaks are reduced to two line breaks. I.e. successive blank lines are reduced to a single blank line
- If after the removal of leading spaces, a line starts with a forward slash, the slash is removed; the only purpose of this arrangement is to be able to force a line to start with some leading spaces after all, or to force multiple successive blank lines.
- The reverse slash character is used as a logical escape, to enable the display of characters with a special meaning (see Appendix B). The reverse slash itself can be escaped, of course, if it needs to be displayed as an ordinary character.
- All unescaped underscores are converted into spaces. This is another mechanism for forcing leading spaces in a line of text.

In-line texts

In-line texts are accepted in all circumstances in which a reference to a named text is acceptable. The basic syntax is extremely simple. If a string beginning with a double quote is encountered where a text name would be appropriate, everything (line feeds included!) up to the next double quote is accepted as the desired text. E.g.

```
say "This is a line of text, follwed by a blank line...

 followed by this line."
```

The **acdc** translator in fact declares a standard A-code text, assigns an automatically generated name to it, and replaces the whole quoted string

with that name. The upshot is that *all* text features described in this section apply to in-line texts as well.

There is an additional syntax wrinkle to permit in-line texts to be text fragments and to permit them to have their own internal dynamics, as described in the dynamic implicit qualifiers section further in this section.

If an in-line text starts with one of **i**, **c**, **r** or **f**, followed the colon character **:**, it gives the text a special property. If the character is **f** ("f:") then the text is deemed to be a fragment, without a trailing line feed. The other three possibilities declare the text dynamic method to be one of *increment*, *cycle* or *random*, as explained in the dynamic explicit qualifiers section below.

Since fragment texts may also have their own dynamic methods, this additional syntax may be repeated as e.g. "f:c:" or "r:f:" etc.

Centered text

A text definition line beginning with the plus character + is treated as a line to be separately centered on the display. The plus sign is stripped off, and the line is prefixed with a line break (unless it happened to be preceded with one) and a line break is appended to it. The result is displayed centered.

A-code also understands "block centering". A centered block is a set of successive text definition lines, each of which is prefixed with the equals character =. The equals signs are stripped off, and the whole block is displayed offset to the right in such a way that its longest line appears centered on the display.

As with individual line centering, the block is prefixed by a line break if one is required (i.e. there wasn't one already), and suffixed with a line-break. Ends of line within the centered block are also honoured, contrary to the more general A-code convention.

Here is an artificial example showing both kinds of centering:

```
TEXT CENTERING.EXAMPLE
   This text shows
```

```
+a centered line
+and another centered line,
 as well as
=a whole block
=of lines, all of which
=are centered as a single unit.
Which is sometimes useful.
```

If displayed (e.g.) in console (i.e. a fixed size font) on an 60-character wide display by

```
say centering.example
```

this would result in

```
------------------------------------------------------------
|  This  text shows                                        |
|                      a centered line                     |
|                   and another centered line,             |
|  as well as                                              |
|                a whole block                             |
|                of lines, all of which                    |
|                are centered as a single unit.            |
|  Which is sometimes useful.                              |
------------------------------------------------------------
```

Non-console mode (i.e. proportional font) displays achieve the same overall effect.

Text post-processing

A-code texts are not displayed to the player immediately. They are accumulated instead in an internal, dynamically sized buffer. The contents of this buffer are post-processed and displayed by the kernel when either a command prompt is issue to the player, or the game is about to exit. This permits some A-code directives to affect previously output text.

For example the **resay** directive, instead of appending its text to the output buffer, first empties the buffer so that its text completely replaces anything accumulated so far. On the other hand the **append** directive preserves the accumulated text, but strips from it any terminating blank lines, effectively appending its text to the last output paragraph.

The A-code kernel processes the accumulated buffer before displaying its contents, e.g. by normalising any successive blank lines to a single blank line. (Multiple blank lines may be magicked up by using the non-breaking space character – see the list of special characters in Appendix B.

A-code text-morphing features

This part of the section describes the "text-morphing" features of A-code texts.

Text switches defined

A text switch is effectively an indexed array of strings embedded in a text message. More than one text switch can be embedded in a single text, and on the other hand, a whole text may consist of a single text switch.

Formally, a text switch is a sequence of arbitrary text strings separated by forward slashes, with the whole sequence being enclosed in square brackets. E.g.

```
[None/One/Two/Many]
```

is an example of a simple text switch of four elements.

The idea is that when the whole message is displayed to the player, just one element from every given switch is selected for display, according to some rule. The selection is made by using some "qualifier" value as the index of each text switch array encountered in the message. Text switch elements are indexed from zero upwards, so that in the above example the element **None** has the index value of zero, while the element **Many** has the index value of three.

While ends of lines are treated in A-code text definitions as white spaces (just like, for example in HTML), an end of line immediately following a text switch separator character / is simply ignored, which conveniently

allows text switches to be spread across several lines. E.g. the above shown switch could have been equivalently written as

```
[None/
 One/
 Two/
 Many]
```

because A-code ignores any line-leading (and line-trailing) spaces in text definitions.

Simple text switches, explicitly qualified

This text morphing feature was introduced by Dave Platt to handle messages such as the report of dwarf attacks on the player. Here's the definition of a text called **knives.thrown** and containing two text switches:

```
TEXT KNIVES.THROWN
    [/One/Two/Three/Four/Five/Six/Seven/Many] nasty sharp
    kni[/fe is/ves are] thrown at you!
```

This is displayed by an A-code text-handling language primitive SAY:

```
say <text_name>, [<explicit_qualifier>]
```

or in this specific case

```
say knives.thrown, thrown
```

The second parameter ("thrown") is in this case the name of a variable holding the number of knives thrown at the player, though from the point of view of the language syntax, it could be a constant, or any A-code entity possessing a value (e.g. an object or a location). In the absence of an implicit qualifier (to be explained later), this value is used to index any text switches embedded in the text supplied as the 1st parameter, where switch elements are counted from zero. So, if one knife is thrown, the displayed message will read:

```
One nasty sharp knife is thrown at you!
```

But suppose more than one knife is thrown, say 5 of them. The first switch is unproblematic -- its index values go from 0 to 8 -- but the second switch only has elements 0, 1 and 2. This is handled by the primary rule of text switches: *use the index value nearest to the qualifier value*. So the message becomes:

```
Five nasty sharp knives are thrown at you!
```

While in this example, the explicit qualifier was supplied as a variable, any A-code entity which has a value associated with it (e.g. an object, or a location, or a plain numeric constant) is also acceptable as an explicit qualifier.

Repeated text switch elements

In practice, text switch elements can be long, multi-line strings, and sometimes it is necessary to have some of the elements repeated. For example, if a player has a purse which can contain some coins, its contents could be described by the following text:

```
TEXT PURSE.CONTAINS
   There [are/is/are]
   [no/one/two/three/several/several/several/several/
   several/several/many] coin[s//s] in the purse.
```

Such repetition is clearly wasteful, as well as tiresome, and the maintenance of switches with repeat elements is unnecessarily complex, because any changes to the repeated elements need to be applied to each of them.

To get around this, A-code interprets a switch element consisting of the single character = to mean a repeat of the immediately preceding element. This in turn may have the same special format, in which case its preceding element is considered. For obvious reasons, the first (zero index) element of a switch may not be a repeat element.

Using this convention, the above PURSE.CONTAINS text could be defined as

```
TEXT PURSE.CONTAINS
   There [are/is/are] [no/one/two/three/several/
```

```
=/=/=/=/=/many] coin[s//s] in the purse.
```

When qualified by an actual number of coins, this would specify the number as *several* for 4 to 9 coins and as *many* for any more than that.

Word and value holders

A-code texts can also contain word and value place-holders, which get replaced dynamically at run-time by words or values specified by the explicit qualifier. Place-holders of either kind can occur both outside and inside text switches.

The dollar sign **$** is used as a value place-holder. If an unescaped dollar sign is encountered in a text to be displayed, and if the text is used with an explicit qualifier, the dollar is replaced with the numerical value of the qualifier. For example, the PURSE.CONTAINS text defined in the last section, could be re-defined as

```
TEXT PURSE.CONTAINS
   There [are/is/are] [no/$] coin[s//s] in the purse.
```

In this case, the A-code statement

```
say purse.contains, 13
```

would result in the display of

```
There are 13 coins in the purse.
```

Of course, as before, any A-code value-bearing entity (variable, location, object...) could be used as a qualifier, instead of a constant value.

A word place-holder is signalled by an unescaped hash sign #, and is conceptually similar, except that a word, rather than a value, indicated by the explicit qualifier is inserted in place of the hash sign. Just what can be used as an explicit qualifier in this case is a bit more complicated.

All declared vocabulary terms obviously have one (or more, if there are synonyms) actual word associated with them. At the author's discretion,

most objects and possibly some locations will also have an associated vocabulary word or words. In all these cases there will be a "primary" word -- the first one declared in the list of synonyms (if there are any synonyms). It is this primary word that gets inserted in place of a word place-holder.

All of this is easier to understand on some practical illustrations. Take for example the object chair1 with the associated nouns "chair" and "seat". Should there be a text declared as

```
TEXT NO.KILL.THINGS
   The # is not something mortal, so cannot be killed!
```

and an object declared as

```
OBJECT -CHAIR1, CHAIR, SEAT
```

where the name CHAIR1 is explicitly excluded from the player's vocabulary, then the statement

```
say no.kill.things, chair1
```

would produce

```
The chair is not something mortal, so cannot be killed!
```

What makes this much more useful than may appear at the first glance, is the fact that A-code variables may store either values **or** pointers (references, to you Algol fans!) to arbitrary A-code entities. Hence if the game code executes somewhere the statement

```
lda target, chair1 # Point variable TARGET at object CHAIR1
```

where "target" is a variable name, then a subsequent statement

```
say no.kill.things, target
```

will produce exactly the same display text as if the object "chair1" or just the simple noun "chair" were used as the explicit qualifier.

Word place-holders really come into their own because the mandatory A-code variables ARG1 and ARG2 hold respectively the verb and the noun of

the player's last command. So assuming that the player said "KILL CHAIR", then

```
say no.kill.things, arg2
```

would once again tell the player that chairs are not for killing.

However, there is a further subtlety here, when it comes to using player command words pointed at by the ARG1 and ARG2 variables, because in this case what is echoed as a part of the response is *not* necessarily the primary word associated with the referenced object, but the word actually used by the player (allowing for expansion of abbreviations, typo correction and vocabulary folding -- see the section on the full 3D structure of the A-code vocabulary). So if the player typed "KILL STO" (note the abbreviation of STOOL to STO, assumed to be unambiguous in this example), the displayed response would be

```
The stool is not something mortal, so cannot be killed!
```

Word place-holders can also appear both within and without text switches, but we'll cover that somewhat later on, under the heading of switch and holder interplay.

Simple implicit qualifiers

Not all texts need explicit qualification. For example, any texts associated with objects or locations (i.e. various forms of object and location descriptions) are deemed to be implicitly qualified by the current state of the object or location -- i.e. by its current internal value.

Take for example the object BATTERIES defined as

```
OBJECT BATTERIES, =BATTERY
   [/Fresh/Worn-out] batteries
  %[There is a pair of brand-new batteries in the goods tray./
   There are fresh batteries here./
   Some worn-out batteries have been discarded nearby.]
  &The two batteries are just the right size and shape for the
   lamp. Both are marked as "[BRAND-NEW/FRESH/WORN-OUT]" in
   chunky [blue/green/red] letters.
```

which has the usual triplet of the inventory, ordinary and detailed descriptions. Each of these contains one or more text switches, which will get automatically qualified by the state of the object. So for instance, if the batteries are spent, the statement

```
describe batteries
```

will display

```
The two batteries are just the right size and shape for the
lamp. Both are marked as "WORN-OUT" in chunky red letters.
```

Similarly the INVENTORY command and the general LOOK command will display their appropriate descriptions qualifying the embedded text switches by the object's state.

It is important to note that implicit qualifiers *always* override any explicit qualifiers -- a point which will have a great significance in the next section. For now it is sufficient to observe that

```
describe batteries, 0
```

would be completely pointless.

Dynamic implicit qualifiers

As already noted elsewhere, A-code texts also carry an internal state (or value), initialised by default (like all A-code values) to zero. Note, however, that to avoid overriding explicit qualifiers in simple text switches, internal text states are only used as implicit qualifiers if a "method" for their manipulation is given as a part of the text definition. Formally, a full text definition looks like this:

```
TEXT [<method>] <text_name>
   <line_of_text>
   [....]
```

where <method> is one of **increment**, **cycle**, **random** and **assigned**. All of these have different effects.

- If the method specified is **increment**, then every time the text is displayed, its internal state is incremented by one, until it reaches the number of elements of the switch with most elements embedded in the text. Here is an example from Adv770:

```
TEXT increment ONCE.IS.ENOUGH

  [And thank goodness for that! /]I [/really /*really* /
  *REALLY* ]don't know why you decided to go and get lost
  in that dark forest. Let's say [once/twice/thrice/
  enough] is enough and not do it again, huh?
```

  This will automatically produce messages of increasing exasperation, as the value of the text tick up after every use, finally sticking at the value of 4.

- The **cycle** method also causes the internal text state to increment by one, but the index value used by each individual switch within the text is the state value modulo the switch size, and the state gets reset to zero once it reaches the least common multiple of all switch sizes within that text.

  This is best demonstrated on a purely artificial example:

```
TEXT cycle DIGITS
   [1/2] [1/2/3] [1/2/3/4]
```

  If used repeatedly, the state of this text will increment by one from zero to 11 and then return to zero again and repeat the same cycle. In the process it will produce successive displays "1 1 1", "2 2 2", "1 3 3", "2 1 4", "1 2 1", "2 3 2", "1 1 3", "2 2 4", "1 3 1", "2 1 2", "1 2 3", "2 3 4", "1 1 1" etc...

  As you can see above, each of the switches cycles independently of the rest, yet only one text state value is driving the whole process. This is very useful in assembling automatically at run-time a wide variety of responses.

- The **random** method does what one would expect. After the text is displayed using its current state value, this value is reset at random to a value within the index range of the largest embedded text switch. The new value chosen is guaranteed to be different from the current one, so

for switches with two components, this method acts exactly as the cycle one.

- The **assigned** method is really a pseudo-method in that it does not actually do anything, other than enabling the text value as an implicit qualifier. Without it, the internal text state is ignored by switch processing, and an explicit qualifier has to be supplied instead.

It should be noted that none of the above described methods, nor the "null" non-method (i.e. when no method is specified) preclude the state value to be assigned into a text or numerically manipulated and examined like any ordinary variable. So for example, taking the above defined DIGITS text, the statements

```
say digits      # Internal state starts from 0!
add digits, 10  # It got incremented to 1, will now be 11
say digits      # It's now 11 will be reset back to zero
say digits
```

would result in "1 1 1", "2 3 4", "1 1 1"

In-line texts can also have dynamic implicit qualifiers. If such a text starts with the colon character ':', followed by a letter, followed by another colon, these three characters are stripped off and a dynamic method is associated with the in-line text in question on the basis of the letter between the two colons: **i** for *increment*, **c** for *cycle*, **r** for *random*. Note that the absence of the *assigned* pseudo-method, on the grounds that there is no way for an in-line text to be referred to by the rest of the A-code source.

Switch and holder interplay

Now that we have been through the details of implicit qualifiers, the interplay between text switches and place-holders can be stated very simply as two rules:

1. Only explicit qualifiers are used when substituting for place-holders of either kind.

2. For processing text switches, implicit qualifiers always override explicit qualifiers.

This enables responses such as exemplified by this Adv770 text:

```
TEXT cycle ITS.JUST.A
    It's just a #. [Nothing very remarkable about it/Not
    remarkable in any way/Nothing special to it/The apt
    description is "unremarkable"].
```

The typical use for this text is

```
   say its.just.a, arg2
```

which will replace the word place-holder in the text with the noun from the player's last command, while cycling in its successive uses through its embedded text switch.

It may at first appear strange that implicit qualifiers are ignored in place-holder substitution, while explicit qualifiers are overridden by implicit ones for switch processing. However, this arrangement does exactly what is often wanted, because it makes it much more sensible to have place-holders within text switches. Again, an example from Adv770 is probably a good illustration:

```
TEXT cycle NOCOMPRENDE.VERB
    [My ignorance shames me, but I do not know what action
    might be signified by "#"./Alas, my vocabulary is too
    limited to encompass "#". Try some other verb?/Very
    remiss of me to be sure, but I've never learned to "#"./
    To my shame, I have no idea what you mean by "#"./"#"?
    Sorry, I don't what it means./I am afraid "#" is not a
    verb I've ever learned./Ahem... "#" is not in my
    dictionary. Would you care to re-phrase?/Regrettably,
    that is not something I know how to do.]
```

If the game fails to make sense of the player's verb, all it has to do now is

```
   say nocomprende.verb, arg1
```

thereby producing a wide variety of responses.

As an aside, in practice I find that the **cycle** method is much more useful for this purpose than the **random** one. Contrary to our intuitive expectation,

randomness tends to be non-uniform (i.e. "clumpy") and hence requires a large number of options, if obvious repetitions are to be avoided.

Text tying

As already noted, A-code variables can be in fact pointers to other A-code entities. The same is true for internal values of A-code texts. A text can be "tied" to another value-bearing entity, thereby removing the need for the game code to explicitly ensure the text value stays in sync with the state of that other entity.

In effect, tying texts to other entities is also a text "method" in that it activates the text's implicit qualifier (which in this case happens to be the value of the entity to which the text is tied). Because this additional text method is not purely internal to the text, there may be reasons for the tying to be performed dynamically within the game code. Hence tying is performed by means of an executable language statement, rather than in text declaration.

Once again, an example may be of help. Adv770 has a quartz seal, which can have one of two states. If the player tries to examine the seal when it is not in his inventory, the game tells him the seal is too small and needs to be picked up. The actual wording of this admonition depends on the state of the seal, hence the game initialisation code contains the statement

```
tie pick.up.seal, seal
```

This effectively ties the value of the text PICK.UP.SEAL to the value of the object SEAL. From now on, if the state of the object changes, the message displayed by

```
say pick.up.seal
```

will automatically change to match, because PICK.UP.SEAL contains a two-component text switch.

Yes, in this particular case one could use SEAL as an explicit qualifier:

```
   say pick.up.seal, seal
```

The seal example merely illustrates the technique. It is too simple to show the justification of that technique. The actual motivation for introducing text tying in A-code was provided by my efforts to resolve a highly complex problem of constructing a location description which depended on several independent factors. It is far too complex to be presented as an illustrative example.

Text nesting

Sometimes a number of separate messages (e.g. location descriptions) consist of a part which is common to them all, and another part which is specific to a given message (or group of messages). This can present maintenance problems (e.g. fixing a typo in *all* identical parts) and is also wasteful. A-code offers an alternative approach. The common message part can be defined as a separate text fragment (i.e. a text without a trailing end-of-line character), which can be "nested" within individual messages.

An A-code message text may include the symbolic name of another text enclosed in unescaped braces (i.e. curly brackets): {}. When the message is displayed, this construct gets replaced with the text indicated by the text name within the braces. The nesting mechanism is recursive, so the nested text may have further texts nested within it.

Here's an example from Adv770, showing the declarations of the first six of the Adv550's ice tunnels. It uses double-level nesting.

```
FRAGMENT INTRICATE.TUNNELS
   You are in an intricate network of ice tunnels.
#
FRAGMENT ICE.DEAD.END
   {INTRICATE.TUNNELS} The only exit is
#
FRAGMENT ICE.TUNNELS
   {INTRICATE.TUNNELS} Exits lead
#
PLACE ICE.CAVE.1
   {ICE.TUNNELS} north and west.
#
PLACE ICE.CAVE.1A
   {ICE.DEAD.END} south.
#
```

```
PLACE ICE.CAVE.2
   {ICE.TUNNELS} north, east and west.
#
PLACE ICE.CAVE.2A
   {ICE.TUNNELS} north and south.
#
PLACE ICE.CAVE.3
   {ICE.TUNNELS} north and east.
#
PLACE ICE.CAVE.3A
   {ICE.DEAD.END} south.
```

Nested texts may have their own implicit qualifiers, of course. If the top level text is used with an explicit qualifier, this is passed on as an explicit qualifier to all nested texts, to any depth.

As a special case, A-code also permits nesting of the ARG1 and ARG2 variables, which are treated as the words actually typed by the player (possibly expanded from an abbreviated state, and typo-corrected). This is used, for example, as follows:

```
   TEXT YOU.DO.IT
      You {ARG1} the {ARG2}.
```

If the player typed (e.g.) "G LAMP", and succeeded in picking up the lamp,

```
   say you.do.it
```

would display "You get the lamp."

Appendix A: A-code text-handling primitives

say <text_name>, [<qualifier>]
Display the named text with an optional explicit qualifier.

resay <text_name>, [<qualifier>]
Like **say**, but completely replaces any text output accumulated since the last prompt.

append <text_name>, [<qualifier>]
Like **say**, but strips off any end-of-line characters at the end of any text output since the last prompt, before displaying the named text.

quip <text_name>, [<qualifier>]
Like **say**, but having displayed the text, aborts all further processing and jumps to the top of the main loop (i.e. equivalent to a **say** immediately followed by a **quit**).

respond <vocabulary_word> [...] <text_name>, [<qualifier>]
Like **quip** but executed conditionally -- only if the player's last command contained any of the listed vocabulary words.

smove <location> <text_name>, [<qualifier>]
An amalgam of **say** and **move**, equivalent to saying the specified text and then moving the player to the nominated location.

vocab {<object>|<place>|<verb>}, {<place>}, {<flag>}, {<text>}

This can be thought of as a wrapper for the SAY directive, which is used to display game's vocabulary in a context sensitive manner (e.g. omitting any nouns referring to objects the player has not yet seen). See the section dealing with A-code vocabulary handling.

Appendix B: Special characters in text definitions

The following characters are have special meaning in text definitions, and have to be escaped with a reverse slash if they are to be displayed "raw".

(reverse slash)
A universal logical escape character. Any immediately following character other, including the reverse slash itself, but excluding end-of-line, is treated as a literal character with no special meaning.

$ (dollar)
A value place-holder, replaced by the current value of the explicit qualifier

(hash)

A word place-holder replaced by the primary word associated with the explicit qualifier.

/ (forward slash)
: If found as the first non-blank character on a line, the forward slash is replaced by a line feed, except if it is the very first non-blank character of the whole text definition, in which case it issimlpy removed. Any blanks immediately following it are not stripped off. If a forward slash is found within a text switch, it delimits text switch components. Not special otherwise.

[and **]** (square brackets)
: Signal the beginning and the end respectively of an embedded text switch.

{ and **}** (curly brackets or braces)
: Enclose the symbolic name of a nested text.

< and > (angle brackets, or less-then and greater-then)
: These enclose HTML tags. All tags are quietly removed in non-HTML modes, echoed as they are otherwise.

+ (plus sign)
: If found as the first non-blank character of a line, signals an individually centered line. Not special otherwise.

= (plus sign)
: If found as the first non-blank character of a line, signals a line of a centered block. If found as the single character constituting a text switch component, represents a back reference to the previous component. Not special otherwise.

_ (underscore or underline)
: A forced blank, not removed by the line-trimming mechanism

Appendix C: Handling anonymous texts (deprecated!)

Anonymous texts can be only handled through pointers. Count text declarations backwards from the anonymous text in question, until you come to a named text. The resulting count is the anonymous text's offset from that named text. Point a variable at that named text and then add the offset to the variable. The variable now points at your anonymous text and all A-code primitives which handle automatic indirection, will accept the variable as a reference to the anonymous text.

Here's an example in the shape of a complete A-code test program:

```
style A-code 12
TEXT FIRST.TEXT
   First text
TEXT
   Second text
TEXT
   Third text
init
   local ptr
   lda ptr, first.text
   add ptr,2
   say ptr
   stop
repeat
```

It will print "Third text" and then stop. (NB: The repeat section is null, but the translator will complain if it is absent.)

Debugging A-code games

Since A-code sources get translated into ANSI C for compiling and linking, debugging can performed on the C level as well as on the A-code level.

C-level debugging

C-level debugging rarely required, but if necessary, can be done using standard debugging tools after building derived C sources into an executable with the -g option. If using gcc it is also useful to add -gdwarf-2 -g3, since that makes GNU debugger gdb understand macro names in the C code.

However, derived C sources are not human-friendly because they use numerical refnos to reference game entities. To assist code comprehension, the *acdc* translator has the -d command line option, which adds to the generated C code printout (on stderr) of A-code source lines being executed. This makes game debugging, be it with with gdb (or similar), or just by visual inspection, much easier.

A-code level debugging

Most A-code debugging takes place on the level of the A-code language itself. While there is no A-code debugger, there are some useful debugging tools.

Runtime procedure call trace

As already noted, *acdc*'s -d command line option has the effect of showing at run-time on stderr A-code lines effectively being executed. This display includes the name of the source file and the line number of the code line being shown. Since the display is on stderr, it can be diverted into a file, regardless of the game's build. In the console mode, where by default both stdout and stderr are shown to the player, game response to a command

comes after this listing of source lines, so that the game is still playble. Thus, for example:

```
 ? out
[...]
repeat.acd:934    ifeq context, none
repeat.acd:935       ifnear door1
repeat.acd:936          and
repeat.acd:937       ifeq waterfall, opened
repeat.acd:938          and
repeat.acd:939       ifeq dwarven, 0
repeat.acd:943       iflt stage, adventuring
repeat.acd:946       input
 You're at end of road again.

 ?
```

Cross-reference lists

The *acdc*translator's -x option makes it emit a cross-reference file .xrf. This file can be further processed by the Perl script *sortxrefs* supplied as a part of the A-code sources package, which read the .xrf file and produces three files, suffixed respectively with .xrefs, .nrefs and .rrefs.

The .nrefs file list game's named entities in alphabetical order, associating each with the refno assigned to it by the translator.

The .rrefs file also lists entity names and their associated refnos, but this time in the refno order.

Finally, the .xrefs file is the most useful one of the lot. It is sorted on entity names and shows where each entity occurs in the source code, differentiating between its declaration and its use. Here is a brief extract from adv770.xrefs (the whole file runs to over 41 thousand lines):

```
aurora.borealis    TXT      7879 text.acd
aurora.borealis    txt     11756 at.acd
 automatic.gate    TXT      2590 text.acd
 automatic.gate    txt       690 actions.acd
 automatic.gate    txt      9155 at.acd
 automatic.gate    txt      9177 at.acd
      available   STATE       597 defs.acd
      available   state     15701 at.acd
      available   state     15714 at.acd
        awarded   STATE       598 defs.acd
        awarded   state     15715 at.acd
        awarded   state     15717 at.acd
        awarded   state      2154 procs.acd
```

```
axe    OBJ       799 objects.acd
axe    obj       702 actions.acd
axe    obj       726 actions.acd
axe    obj       778 actions.acd
...    ...           ...
```

So, for example, the object AXE is declared (type is in capitals!) on line 799 in the file objects.acd and referenced (type in lower case) in lines 799, 702, 726, 778... in the file actions.acd.

Game data dumps

The minor directive DUMPDATA, dumps the current state of the game to standard error. The default display contains no symbolic names (because they are not known to the kernel), so interpreting it requires constant reference to the above mentioned cross-reference files. This is not at all convenient.

However, if the game is translated int C with the -d option, or if its source code defines the special variable ENTNAME, entity names are passed on to the kernel and are used in the dump. Here are some fragments of such a dump:

```
================= OBJECTS =================
....
   5 it               0    1000000000000000 at    0
   6 keys             0    1001000000000000 at   65 building
   7 lamp             0    1001000000000000 at   65 building
....
================= PLACES =================
  63 road             0    0101110000000000
  64 hill             0    0101000000000000
....
================= WORDS =================
 192 again
 193 carry
 194 drop
....
================= VARIABLES =================
....
 322 penalties         0    0000000000000000
 323 here             63 => road
 324 there            63 => road
 325 status            1    0100000000000000
....
================= TEXTS =================
 502 intro             0
 503 typo              0 cyclic
....
```

```
 693 plant.2           0 tied to   27 plant
...
```

For each entity, the dump gives its refno, its name and (if appropriate) its value, followed by (if appropriate) its properties bit screen (a.k.a. flags). For objects, their location is given (as location refno and name, if any). If a variable is a pointer to another entity, the entity pointed to is shown (its refno is the variable's value). Finally, if a text has morphing features, it's type is shown and, for typed texts, the entity to which they are tied.

By default all of the game's data is dumped, but one can choose to dump only data pertaining to a particular type of vaue-bearing entity by adding OBJECT, PLACE (or LOCATION), VARIABLE (or VARS) or TEXT as an argument to DUMPDATA.

Game data is dumped either to stderr or, if the game is being logged (i.e. was invoked with the -l command line option), to the log file.

As the simplest possible use, you can define DUMPDATA as a verb with a corresponding action:

```
verb dumpdata
action dumpdata
  dumpdata
  quit
```

Or, more sensibly, you can make DUMP one of optional "wizard" actions (see below).

The CHECKPOINT minor directive

CHECKPOINT minor directive is another debugging tool. When executed, it reports its own location (file name and line number) in the A-code source.

```
 ? n

=== Checkpoint: procs.acd, line 37 ===
 You are at the end of the road again.

 ?
```

What makes this useful is the fact that changing executable A-code source has no effect on its ability to restore saved games. Thus it is safe to add CHECKPOINT statements in a suspect piece of code in order to track problems in a saved game.

Constructing non-integral "wizard" mode

A-code's unusual feature of procedure groups permits construction of debugging commands (a.k.a. the wizard mode) as an optional add-on by using the INCLUDE? major directive. Once again this is made more useful by upward compatibility of saved games.

While A-code permits entities being used before being declared, it is useful to place all executable code after all declarative code. (This does not, of course, preclude entities being used before their declarations, since A-code texts can embed references to game entities.) If that's how code is arranged, adding an optional source file (via the INCLUDE? major directive) in between declarations and executable code has two interesting effects:

- It has no effect on compatibility with games saved by the same code, but without the optional file. (See the section on upward game compatibility for an explanation). Note, however, that the reverse does **not** apply. If the optional code defines any additional entities (objects, places, variables or morphing texts), games saved by a version with that optional code cannot be loaded by the version without it.
- The INCLUDE? major directive allows procedures (including REPEAT, ACTION and AT ones) to pre-empt procedures of the same name within the rest of the game's code. Such an intercepting "wizard" procedure can do its stuff and (a) QUIT, terminating the current command loop, (b) RETURN, terminating execution of the procedure group of that name (i.e. skipping the rest of the so-named procedures, or (c) PROCEED, letting the rest of the procedure group execute as it would do without the optional code.

For example, suppose the game contains

```
verb find
action find
```

```
  quip "You'll have to find thing out for yourself. "
```

Suppose further that the optional code contains

```
variable entname
action find
   ifeq status, 2  # Player trying to find something
      ifflag arg2, object
         say "f:Object {ARG2} "
         locate entname, arg2
         quip "is at {ENTNAME}." # Terminates command loop
      else
         quip "{ARG2} is not an object!"
      fin
   fin
```

This would modify the behaviour of the command FIND to show the location of of a nominated object, or, if no object nominated, to do exactly what it would do without the optional code. For ease of debugging, optionally included code should also have commands to toggle "wizard" mode on and off and other wizard mode commands whould simply PROCEED if the wizard mode is off.

There can, of course, be more than one optionally included file, positioned at various places in the source code, as appropriate. Nor is it necessarily the case that such includes must come before other code. E.g. my A-code port of Adv350 (written in order to experiment with mobile NPCs) has an extensive set of "wizard" tools, which is included **after** the NPC movement and actions code. It defines its own vocabulary list, that can be displayed by VOCABULARY WIZARD:

```
 ? voc wiz

 Wizard (i.e. debug) commands:

   close cave                - triggers the next cave closure stage
   decrement <entity>        - decrement state value of object or location
   data [obj|loc|var|text]   - show game's data
   fetch <object> (obtain)   - fetch the object from wherever
   find <object>             - go where the object is
   first                     - forces first dwarf, if axe not seen
   fly <location> (teleport) - go to the named location
   glow                      - toggle magical illumination
   increment <entity>        - increment state value of object or location
   next                      - go to the next higher location
   notbeen                   - show locations not yet visited
   previous                  - go to the next lower location
   runout                    - sets event clock to zero
   [show] {numbers|npcs}     - toggles loc number/npcs repeated display
   show <entity>             - Displays entity's current value
```

```
   where                    - shows where one is (and came from)
   where treasure           - shows valued objects (sorted on seen flag)
   where water              - shows water-holes
   where <object>           - shows object's location
   wizard {on|off}          - switches wizard mode on or off

 ?
```

The optional include file debug.acd can be found in the A-code source of Adv350 available at https://mipmip.org/adv350 in the file opt/debug.acd.

If using the *advbld* script to build Adv350 (or Adv770), the -W script option copies opt/debug.acd to where it will be found by an optional include, builds the game and then deletes the file copy.

A-code language history (as I recall it)

The main purpose of this document is to explain (in response to some requests) why some aspects of my A-code implementation are the way they are, and what the future might hold if I get around to it.

A-code is the language developed by Dave Platt in order to write his influential Adv550 expansion of the classic game Adventure by Crowther and Woods (Adv350 in the modern nomenclature). Originally written in PL6, a Fortran 77 implementation of the original A-code engine was distributed in the late 1980s together with the the Adv550 A-code source.

Dave Platt's A-code engine had a "munger" and an "executive", both written in F77. The munger took the A-code source and produced a tokenised pseudo-binary. The executive was was effectively a virtual machine which executed the pseudo-binary. The lightly encoded game's text was in a separate file.

I first made use of this original A-code implementation to merge Luckett's and Pike's Adventure II (now known as Adv440) with Adv550 and Platt's Adv550 into Adventure4, which later evolved into Adventure4+ (now known as Adv660).

The initial Adventure4 and then Adventure4+ (1983 - 1985/6, on Primes) implementation took the A-code architecture as it was. The only changes to the munger/executive were in improving the command parser (e.g. automatically allowing all words to be abbreviated to the minimal unambiguous length, chaining commands on one line with semicolons, and providing AGAIN for command (simple or compound) repetition.

In September 1990 I embarked on re-implementing A-code in C on Unix, the main aim being to teach myself C (my previous expertise was first in Algol/Algol68, then in Fortran4, then in Fortran77). I later joked that the new version was dedicated to the proposition that a real programmer can write Fortran in any language. That implementation was the seed of the current version and it moved away from Platt's in one crucial step, the full

significance of which did not transpire until much later. Instead of replicating Platt's virtual machine approach, I wrote acdc, which translated A-code directly into compilable C, with game-independent kernel C source providing a library of standard calls used by the translated code.

The reason for this shift was performance. Adventure4+ by that time became too large and complex, yet I wanted it to run on ordinary PCs as they then were. Virtual machine interpreter of pseudo-binary struggled with that, but compiled C worked just fine.

To accommodate minuscule (by modern standards) memories standard at the time and the slow speed of disk access, I nicked from Prime an outline of their paging algorithm, and built that into the kernel, together with the ability to report locate demands, the number of locate buffers being specifiable at compilation time. This mechanism is still present within the kernel, and is required to build DOS versions of games. The kernel also provided the option of reading all texts from the data file as required, with no paging mechanism, or to load the data file entirely into memory on startup. Initially there was no option for pre-loading all of the text into memory -- compilers of the time tended to choke on that.

Unix (Irix in fact) and later Linux became the default platform. Initially I built DOS/Windows executables by using djgpp under DOS. Later DOS and Windows builds diverged. The DOS version is now created using djgpp (run under wine), whereas the Windows version gets built using MinGW and packaged using InnoSetup. At some stage a Mac build was added as well.

As memory capacities grew and general machine performance improved, I first added the option to build the whole game as a single executable with no data file – all text begin stored in initialised arrays in the C source. Later this became the default. Thus things stood until I embarked on Adv770, which is when I discovered some unexpected upsides and downsides of the approach I'd taken.

I needed the game to be tested by others, but I knew from experience that it would be foolish to rely on testers to report all problems. Without knowing what is supposed to happen, it is not necessarily obvious whether things are

going wrong or not. I simply had to have access to testers' log files. The only way this could be achieved was to run the game through a web interface – in the cloud, as one would say these days. Unfortunately, as the acdc/kernel implementation of A-code stood at the time, this simply was not possible. An A-code game was simply an executable, taking player commands from standard input and responding on standard output in a continuous loop.

So I sat about to implement a single-turn operation mode. In this mode, instead of waiting for play command, the game would automatically dump its current state to disk and exit. And when a new command arrived via a web interface, the game would be restarted and the saved state automatically loaded before processing the command. This would be easy enough with Dave Platt's original virtual machine approach, but my translate-compile-and-run implementation entailed that in an A-code game there could be only a single place where the player could be asked for input. Specifically, any use of the QUERY directive, used in Adv550 and Adv660 to handle yes/no queries, was simply out.

My eventual solution was to implement the context mechanism and replace any use of query with setting the special variable CONTEXT to a unique value, specifying the nature of the query, before saving the game state and exiting. Once a response arrived from the player, the game would restart and if the value of the restored CONTEXT indicated that a yes/no question had been asked, it would evaluate the answer in the same way as it would have done if QUERY had been used. This permitted cloud-based operation (initially CGI, later PHP). It also, quite unintentionally, enabled all A-code games (even Dave Platt's original code of Adv550) to have a persistent state. If a game was simply interrupted (or... erm.. crashed), it could be restarted from that point without being explicitly saved by the player.

Some internal changes had to be introduced to support this development, and so the major version number of the acdc and the A-code kernel got incremented from 10 to 11. Version 11 also brought in another innovation. All earlier versions assumed games being played in a terminal emulator, and assumed the display to be limited to 80 characters per line and 24 lines

per screenful. Version 11 added the ability to run a game in its own window, initially by using the GLK library.

On the plus side, since there was no virtual machine to save the state of, and I'd avoided using the QUERY directive, it was pretty easy to maintain upward compatibility with testers' saved games, despite the game undergoing some major bug fixing.

Adv770 was finally released in 2003, but I continued tinkering with my A-code implementation anyway. 2008 brought version 12 of A-code, which removed the requirement of declaring game's entities before they could be used. This was achieved by changing the acdc translator to make two passes over game source instead of one &ndash the first pass collected information on all game entities being declared by the source and the second pass actually translated the source into C.

In changing to version 12 I also took the opportunity of ditching GLK because its Unix/Linux implementation of GLK had severe limitations (and used really ugly, hard to change fonts). Instead an A-code game executable could launch the default browser and then act as a very simple HTTP server, using the browser as the user interface.

In 2013 Brian Ball asked for changes which would enable him to port Adv770 to iOS. This was an interesting challenge, since iOS demanded to be in charge of the game's command loop. So the kernel was twisted yet again, to allow a "library mode". Fortunately, this was not too hard to do, since the game persistent mechanism introduced for the CGI/PHP operation could be adapted for the purpose. The main difference being in returning accumulated text to the calling routine, instead of displaying it to the player.

This change had a large unexpected payoff a year later, when I used emscripten to translate the C-code generated by acdc into JavaScript, so that it became possible to run the game entirely in an HTML 5 compliant browser.

Finally, in 2020 (doesn't time fly!) I got around to making my implementation of A-code to be entirely UTF8 compliant. While UTF8 encoding could always be used in game texts and object/place descriptions,

the challenge was to permit UTF8 in entity names and hence in player vocabulary. That done, I also extended non-vocabulary names convention to vocabulary word declarations which frees A-code games from any dependence on the English language by allowing one to safely re-define the default command parsing words AND, THEN and AGAIN. So if you want to write IF games in Japanese or Russian, you can! :-)

More time passed and AI coding abilities have massively – so much so that Claude was able to fix the QUERY problem. as of A-code version 12.92, the QUERY directive now works in the single turn mode too. So do multiple INPUT commands. Hence the CONTEXT mechanism is no longer required to emulate QUERY if one wants to build an HTML/JS version, or run an A-code game in the cloud.

And that's where things stand at the time of writing.

Contents

Introduction to A-code styles

The notion of A-code styles came relatively late, when I decided in 1990 to teach myself C by re-implementing the A-code engine using the translate/compile architecture, in place of Platt's original munge/interpret one. I wanted the new implementation to support both Platt's Adv550 and my own Adv660 (a merger of Luckett's and Pike's Adv440 with Adv550). Because there were some incompatibilities between A-code of Adv550 and that of Adv660 (most notably text switch components being counted from 1 or from 0 respectively), it became necessary for a game's code to signal how it is to be treated. Logically enough, Platt's A-code was designated as Style 1. Remembering the many changes my form of Platt's munger/executive underwent before Adv660 was made generally available on the Net, I arbitrarily assigned Style 10 to the final version of Adv660. (Luckily it was not Style 2, because years later it was convenient to assign Style 2 to the re-discovered Goetz's Adv580.) Hence the STYLE major directive, to fix the style of an A-code game. As things stand, style numbers are assigned as follows:

- Style 1 – the style of Platt's A-code source of Adv550.
- Style 2 – the style of Goetz's A-code source of Adv580.
- Styles 3 to 9 – lost in the mists of time, having existed briefly in the process of my merging of Adv440 and Adv550 into Adv660.
- Style 10 – the style of my A-code source of Adv660.
- Style 11 – the initial style of my Adv770.
- Style 12 – the current A-code style.

Taking Platt's A-code as the base, let's look at changes brought in by different styles, bearing in mind that only the style 1 to style 10 transition is of historical significance.

From Style 1 to Style 2

The style of Goetz's Adv580 A-code differs very little from Style 1.

- It relaxes in-line comment convention. While Style 1 in-line comments must start with the open brace character '{', Style 2 also permits square '[' and round '(' brackets as in-line comment delimiters.
- It introduces CIF and CENDIF major directives in order to use selectively normal or "bowdlerised" versions of some texts.

From Style 1 to Style 10

Text switch modification and generalisation:
Switch components are counted from zero rather than from one.
Switches are permitted in all texts, including object and place descriptions.
Multiple long object/place descriptions (%) deprecated - replaced by text switches.

Changes to major directives:
Deprecated LIST, NOLIST and XREF.
Added NOISE (and preferred) as a synonym to NULLWORD.
Added PROC (and preferred) as a synonym to LABEL.
Added FRAGMENT.
Deprecated SYNONYM in favour of OBJECT name list.

Changes to minor directives: Added DOALL and FLUSH for handling GET/DROP ALL.
Added IFHERE, IFINRANGE and IFIS.
Added ITERATE, QUIP, NEGATE, CHOOSE and RANDOMISE.
Deprecated ITLIST (synonym of ITOBJ).
Deprecated AT, HAVE and NEAR.
Deprecated EOF, EOI, EOR (subsumed into FIN).
Deprecated NAME and VALUE (subsumed into SAY).

Player interface
Allowed compound commands.
All player vocabulary words automatically abbreviated to the minimal unambiguous length.

GET/DROP ALL can be enabled by game code.
AGAIN can be enabled by game code. Object detailed description category (&).

From Style 10 to Style 11/12

Since enhancements that came in under style 12 are also retrospectively available in style 11, it makes sense to lump 11 and 12 ogether for the purposes of compaing the to style 10. The change from 11 to 12 was dictated by a major surgery on the *acdc* A-code to C translator to make it operate in two passes instead of one. The pupose was to allow game entities to be referred to by game code before the relevant entity declarations. This is particularlly handy in debugging via "wizard mode" code optionally included in game source via the INCLUDE? major directive. See the section on debugging A-code games.

Changes to major directives:
Added ARRAY, STATE, CONSTANT, FLAGS

Canges to minor directives:
Added FAKEARG and FAKECOM
Added OTHERWISE
Added IFLE, IFGE and IFNE
Added IFHTML, IFTURN and IFCGI
Added IFDOALL and IFTYPED
Added LOCAL
Added UNDO and REDO
Added RESAY and TIE
Added RESPOND
Added SAVE and RESTORE
Added INTERSECT
Added VERBATIM and VOCAB
Deprecated BIT, BIS, BIC in favour of IFFLAG, FLAG and UNFLAG respectively

Deprecated KEYWORD, ANYOF and NEAR in favour if IFKEY, IFANY and IFNEAR

Further functionality added:

- ITOBJ takes flag and state supplementary arguments.
- Multiple args for IFAT, IFIS, IFLOC
- Multiple args for FLAG, UNFAG.
- In-line texts.
- CGI mode.
- Library mode.
- Local variables, and procedure arguments.
- Context and undo mechanisms.
- Typo correction, output word scanning.
- UTF8 support.
