

THE ABC NEWSLETTER

ISSN 0922-8055

CWI, Amsterdam

Issue 7, Feb 1990

CONTENTS

New Publications
Some Notes
How To Get and Install Your Copy of ABC
How To Install ABC on Unix Machines
ABC Manual
A Short Introduction to the ABC Language
An Adventure Program
A Histogram Program
How To Order ABC

Announcing:
the ABC Release, for

Atari ST
Apple Macintosh
IBM PC and compatibles
Unix

THE ABC NEWSLETTER

New Publications

The ABC Programmer's Handbook is at last available.

The ABC Programmer's Handbook.

Leo Geurts, Lambert Meertens and Steven Pemberton, 164 pages.

The book is an introduction to ABC and its implementations, for people who have already programmed.

It consists of first a quick introduction, giving a brief and informal overview of the whole language. This is followed by a chapter of example programs. These examples demonstrate ABC style, and how you program some common data structures in the language. After that is a user's guide to using the ABC implementations, explaining how to use the ABC system, how to manage ABC programs, and so on. Finally, there is the Description of ABC, a semi-formal definition of the language, giving the syntax and semantics of the language, and the definition of the built-in commands and functions. Appendices give a Quick Reference, and details of the implementations.

Published by Prentice-Hall, Englewood Cliffs, New Jersey, 1989, ISBN 0-13-000027-2.

Some Notes

The Newsletter is intended to provide information about ABC and to provide a forum for discussions.

This special issue is mainly devoted to the release of ABC. It contains various documents concerning and accompanying this release.

Write to us if you want to be added to our mailing list.

You are encouraged to submit any articles you see fit. Articles don't have to contain fully thought-out ideas, but may be yet undeveloped thoughts intended to stimulate discussion. The kinds of articles we have in mind are: interesting programs, either written or suggestions; unusual applications; letters, discussions on points of the language, proposed improvements, experience with the language, and so on.

If you are fortunate enough to be connected to a network with a gateway to UUCP net, you can submit articles and send mail to:

`abc@cw.nl`

For older mailers, use:

`... hp4nl!cw.nl!abc`

Otherwise, articles and mail should be sent to

The ABC Newsletter
CWI/AA
POB 4079
1009 AB Amsterdam
The Netherlands

How To Get and Install Your Copy of ABC

Most of this newsletter is dedicated to the new release of ABC. This first article explains how to get hold of your version of ABC, and how to unpack or install it.

There are four versions of ABC: for Atari ST machines, for Apple Macintoshes, for IBM PC's and compatibles running MS DOS, and for Unix machines. ABC programs and workspaces can be transported between all these machines without any change.

All versions contain a version of the ABC Manual in the file "abc.hlp", that can be consulted from ABC by pressing the ?-key at the command prompt (the Macintosh version has a Help menu instead). A short introduction to ABC is also included in the file "abcintro.doc" ("abc/doc/abcintro" for the Unix version). These documents are printed in this newsletter. Consult them to learn how to use ABC on your machine.

If you have a personal computer with a hard disk, the best place to install the ABC files from the distribution is in a directory (or folder) "abc" at the root.

Get Your Copy

If you can, try to get copy of ABC from someone you know.

The next best way is to log in by anonymous ftp on "mcsun.eu.net" ("-n 192.16.202.1"). You will find all versions of ABC in the directory "/programming/languages/abc":

```
ftp mcsun.eu.net
Connected to ...
Name: anonymous
Password: ...
ftp> cd /programming/languages
ftp> cd abc
ftp> get README -
```

The last command will show you the contents of the file "README", which will tell you under which filenames the different versions of ABC are available.

Since all versions will have been posted to the applicable newsgroups on USENET (e.g. comp.sources.unix, comp.binaries.ibm.pc, comp.binaries.mac and comp.binaries.atari.st), there will be various other archive machines and bulletin boards that carry versions of ABC.

If this does not succeed, and you are connected to a network with a gateway to UUCP net,

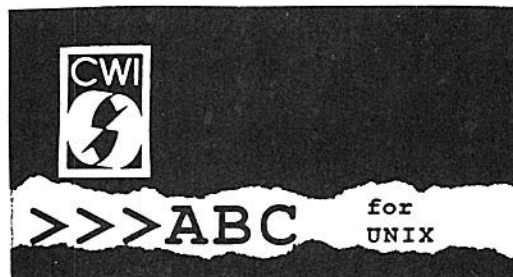
you can try to send a message to our mail-server at "info-server@hp4nl.nluug.nl" containing the lines:

```
request: index
topic: index
```

This will give you a short introduction on the use of this service and a list of available subjects.

Finally, as a last resort, fill in the Order Form at the end of this newsletter.

Send a message to "abc@cwi.nl", if you encounter any problems in getting your copy of ABC.



The Unix Version

The Unix version is distributed in source form.

The ftp version is in the binary image file "abc.unix.tar.Z". Put this in the parent-directory where you want the ABC source to reside, (e.g. "/usr/local/src"), and type:

```
zcat abc.tar.Z | tar xf -
```

(sometimes you need to type "xof" instead of "xf" to make yourself the owner of the files extracted from the tar-file; see "man 1 tar").

All files will then be unpacked in the subdirectory abc (e.g. "/usr/local/src/abc/").

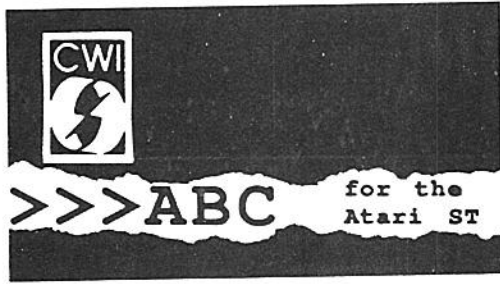
If you have ordered a tape, mount it and type something like:

```
cd /usr/src/local
tar x      # or tar xo
```

which will extract all files into the directory "/usr/src/local/abc", where you can start the installation.

Consult "How To Install ABC on Unix Machines", later in this newsletter to find out how to install ABC on Unix machines. You can find this documentation on line in the files "abc/README", "abc/README2" and "abc/Problems".

About 1.4 megabytes are needed for the source. You need 2.5 megabytes in total to compile and install the system.

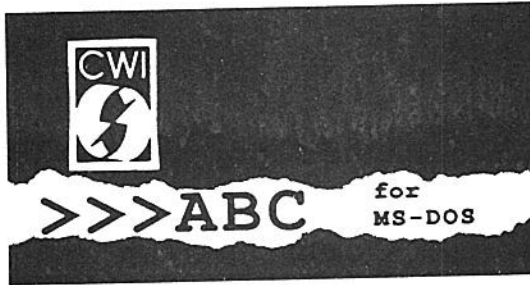


The Atari ST Version

The version for Atari ST machines is in the binary image archive file "abcst.arc". To extract the six files "abc.tos", "abckey.tos", "abc.hlp", "abc.msg", "abcintro.doc" and "readme.st" type:

```
arc x abcst.arc
```

The extracted files total about 373 K disk space.



The IBM PC Version

The version for the IBM PC and compatibles running MS DOS is distributed as a binary image file "abcpc.exe"; it contains a self-extracting compressed archive of five files: "abc.exe", "abckey.exe", "abc.hlp", "abc.msg" and "abcintro.doc". To extract these, change to the disk and directory where you want them and type e.g.:

```
a:abcpc
```

if you saved "abcpc.exe" on a floppy in drive "a:". You can extract the files one by one by typing e.g.:

```
a:abcpc filename
```

(On a one-floppy system type:

```
b:abcpc abc.exe
```

to save the main program on floppy 1. You will be asked to swap floppies by MSDOS. Next type:

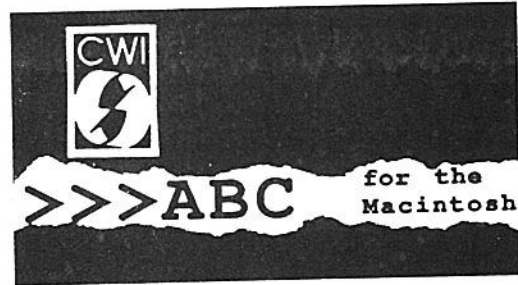
```
b:abcpc abc.msg abc.hlp
abckey.exe abcintro.doc
```

to save the auxiliary files on floppy 2.

To run abc, type:

```
b:abc
```

and use floppy 1 when MSDOS asks for drive b:, and floppy 2 for drive a:. Your workspaces will be saved on floppy 2. On floppy 2 you only need the files abc.msg and abc.hlp. If you created a file abc.key with abckey it must be on floppy 2 too.) All extracted files together use about 410 K disk space.



The Apple Macintosh Version

The ftp version for the Apple Macintosh is in the ascii file "abc.mac.sit.hqx". It is a binhexed StuffIt archive. First choose "Decode Binhex File ..." from StuffIt's Other menu. (Or BinHex itself if you have an old version of StuffIt). Then "Open Archive ..." "abc.mac.sit" from the File menu to extract the four files "MacABC", "MacABC.help", "MacABC.doc" and "ABCIntro.doc". The extracted files comprise about 333 K.

MacABC even runs on my Lisa using MacWorks at home. But on a single-sided, non-hierarchical floppy (400 K), you can only use the whole disk as a single workspace; to get multiple workspaces in ABC copy the files to a hierarchical disk.

* * *

The above information for all versions relates only to the files you receive from ftp. If you get the release on floppy, you won't have to extract the files from an archive yourself but find them directly on the floppy. The only exception to this is the PC version which doesn't fit on a single 5 1/2" 360K floppy unpacked; therefore we send the self-extracting archive "abcpc.exe" and you will have to follow the guidelines above.

How to Install ABC on Unix Machines

You will need 2.5 megabytes in total to compile and load the ABC system. To start type:

```
./Setup
```

which will ask you some questions to set the ABC system up on your installation. You can call "Setup" any number of times without spoiling files. So run it once to see what questions you will be asked. If you don't know the answer to some question, you can run it again.

```
make makefiles
make depend
```

will construct simple makefiles and makefiles with dependencies, respectively, in the relevant subdirectories.

```
make all
```

will compile and load the ABC system, producing the "abc" command file and the binaries and datafiles it needs locally in this directory. You can test the ABC interpreter with:

```
make examples
```

This runs some examples in ./ex. It does not test the ABC editor, however; that can only be done interactively with

```
make try_editor
```

Consult the section "HOW TO TRY THE ABC EDITOR" below (also online in abc/README2). If all is well and you want to make ABC public

```
make install
```

will do some editing to get the right pathnames in, and install the "abc" and "abckey" binary files, the "abc.1" manual file, and the auxiliary files in the directories you indicated during setup. Finally

```
make clean
```

will clean all intermediate object files from the source directories, and

```
make clobber
```

will clean the automatically created makefiles from the subdirectories.

If you have made ABC public, all necessary files have been copied to the public places, and you can get rid of the entire ABC file system hierarchy you extracted from the tape, if you want.

If there are any problems, don't panic. See ./Problems for advice. Also edit the example Bug Report form in ./doc to communicate the problem to us. We can then send out diffs for fixed problems in the future.

Above all, we would be very grateful to receive any comments you have about the setup procedure, or the ABC system in general, on how to make it easier to use.

Good luck!

HOW TO TRY THE ABC EDITOR

The directory ./ex/try is here to try the ABC editor interactively.

After a successful "make all" or "make examples" say:

```
make try_editor
```

to enter the ABC system. (If you are cross-compiling, copy ./ex recursively to the target machine and say:

```
cd ex
TryEditor
```

you should make sure that the "abc" command in your PATH.)

After the ABC system has started up it will prompt for a command with

```
>>> ?
```

Slowly type 's', then 't', (no capitals needed) and you should see the ABC editor suggest the SELECT and START commands, respectively. Now press [TAB] to accept this last suggestion, and [RETURN] to enter the START command to the ABC interpreter. This command will prompt you for input, with

```
?
```

Just enter a few lines of text, (which will be echoed), ending with an empty one (press [RETURN] immediately). A short "poem" should be generated by the ABC interpreter.

If you are already familiar with the ABC language, you might try to edit the START how-to by answering

```
>>> ?
```

with

```
:START
```

For example, try to remove the SET RANDOM command, to get random results on the same input. Or make the unit delay the echoing of the text, entered by the user, until after the reading of the empty line. For testing purposes you should at least try the arrow keys to move the focus around.

You can undo any change by pressing [BACKSPACE].

You can get help with ? or [ESC]-?.

You can leave the ABC editor with [ESC]-[ESC].

You can leave "abc" by typing QUIT.

For more information, see the manual page "abc(1)" and "The ABC Programmer's Handbook". See the B Newsletter, issue 2, for a description of the "generate" program.

HOW TO TACKLE PROBLEMS DURING ABC INSTALLATION

This section contains some detailed advice in case you run into problems while installing the ABC system.

The Setup Procedure

Your best bet if the "Setup" script fails is to read it, locate the problem, change it and run it again. You can always shorten its runtime by changing long pieces into the simple setting of a shell variable. For instance, once you are sure your floating point is alright, you might replace the whole section titled "Floating point arithmetic ok?" by a simple "flag=".

Normally you should not edit the files that Setup creates (./Makefile, ./uhdrs/os.h, ./unix/abc.sh and ./scripts/mkdep) directly, but their ancestors (./Makefile.unix, ./uhdrs/os.h.gen, ./unix/abc.sh.gen and ./scripts/mkdep.gen, respectively) and run Setup to incorporate your changes. If you really want to change them directly, also change Setup to work on them or remove Setup completely.

When "make makefiles" or "make depend" fail

When "make makefiles" fails to create the makefiles */Mf in the relevant subdirectories, first try to edit the shell commands in Makefile.unix (and run Setup again; see above).

Likewise, if "make depend" fails to create the */Dep files in the subdirectories, try to fix ./scripts/mkdep (and incorporate the changes in ./scripts/mkdep.gen before running Setup again).

However, if either of these is not successful, you can use the already constructed makefiles */MF and */DEP. To do this, redefine "MF=Mf" to "MF=MF" and "DEP=Dep" to "DEP=DEP" in Makefile.unix. You can then call "make all" immediately, without "make makefiles" and "make depend".

The makefiles */MF and */DEP were created on a machine running 4.3 BSD UNIX. The dependencies in the */DEP files on system include files (embedded in <>) were stripped to make them more portable. On a different system the real dependencies may differ in some details, however. This may cause a second "make" after some editing to not see all dependencies on include files properly. You can always use "make clean all" to force all objects to be recompiled if you suspect you ran into this.

Machine Configuration

The file `./uhdrs/config.h` is created by compiling `mkconfig.c` and running `mkconfig` on your target machine, since it tries to establish some facts about the hardware configuration. (If you are cross-compiling you should do that before `make depend` since that would run `mkconfig` on the local (compiling) machine. If Setup went alright, `DESTROOT` will be set in the Makefile and you will be warned accordingly.)

If you really have to edit `uhdrs/config.h`, you should edit the Makefile (or `Makefile.unix`) so that it will not overwrite it anymore.

The problem most encountered with `mkconfig` is `unexpected over/underflow`. This is usually caused by a bug in `printf`, where it can't print very large or very small numbers. Look at the last line produced by `mkconfig` before it failed, and then locate the `printf` after the one that printed that line. If it is trying to print a comment (rather than a `#define`), you can safely comment out the `printf` and try again. (You might also want to report the bug to your UNIX supplier.)

Other Unix's

The installation of the ABC system has been tested under 4.3 BSD UNIX on Tahoe, Vax and Sun machines, under ATT System V Release 3.0 UNIX on an Intel 80386, and under MINIX, which is supposed to be VERSION 7 UNIX compatible. The Setup script tries to find out whether your UNIX is one of these, and creates `./uhdrs/os.h` from `./uhdrs/os.h.gen` accordingly. We expect you will have no problems compiling the ABC system in this case.

If your UNIX is different, the Setup script will create a file `./uhdrs/os.h` with most defaults setup for a VERSION 7 UNIX system, since that makes a minimum number of assumptions. Examine the resulting file, and change the names of system include files if they are different on your system. Also check the definitions and UNIX specific flags in this file. See the comments, and use your systems manual to find out how to set them. Don't forget that this file is created by running Setup; change Setup if you want to edit `uhdrs/os.h` directly, or edit `uhdrs/os.h.gen` and run Setup again.

If your machine's memory is not that big, you might examine `./uhdrs/feat.h` to turn off some features in an attempt to make the ABC editor-interpreter smaller.

We have tried to gather the operating system dependent parts in `./unix/*.c` and `./uhdrs/*.h`. Examine these if any problems in compilation remain.

Editor Problems

Once the ABC system is compiled, you may encounter problems when you use the ABC editor. Our experience is that most of these problems are caused by erroneous or insufficiently detailed termcap entries, which describe your terminal's capabilities; so first check the `termcap(5)` manual entry (or `terminfo(4)` for terminfo systems). Ask your system's guru to give you a hand if you are not familiar with these.

We use the following entries from the termcap database if they are defined for your terminal:

Name	Type	Description
AL	str	add <i>n</i> new blank lines
CM	str	screen-relative cursor motion
DL	str	delete <i>n</i> lines
al	str	add new blank line
am	bool	has automatic margins
bc	str	backspace character
bs	bool	terminal can backspace
cd	str	clear to end of display
ce	str	clear to end of line
cl	str	cursor home and clear screen
cm	str	cursor motion
co	num	number of columns in a line
cp	str	cursor position sense reply
cr	str	carriage return
cs	str	change scrolling region
da	bool	display may be retained above screen
db	bool	display may be retained below screen

dc	str	delete character
dl	str	delete line
dm	str	enter delete mode
do	str	cursor down one line
ed	str	end delete mode
ei	str	end insert mode
hc	bool	hardcopy terminal
ho	str	cursor home
ic	str	insert character (if necessary; may pad)
im	str	enter insert mode
in	bool	not save to have null chars on the screen
ke	str	keypad mode end
ks	str	keypad mode start
le	str	cursor left
li	num	number of lines on screen
mi	bool	move safely in insert (and delete?) mode
ms	bool	move safely in standout mode
nd	str	cursor right (non-destructive space)
nl	str	newline
pc	str	pad character
se	str	end standout mode
sf	str	scroll text up (from bottom of region)
sg	num	number of garbage characters left by so or se (default 0)
so	str	begin standout mode
sp	str	sense cursor position
sr	str	scroll text down (from top of region)
te	str	end termcap
ti	str	start termcap
ue	str	end underscore mode
up	str	cursor up
us	str	start underscore mode
vb	str	visible bell
ve	str	make cursor visible again
vi	str	make cursor invisible
xn	bool	newline ignored after 80 cols (VT100 / Concept glitch)
xs	bool	standout not erased by overwriting

Of these your termcap entry should at least define the following:

```
le OR bc OR bs
up
cm OR CM OR (ho AND do AND nd)
(al AND dl) OR (cs AND sr)
ce
(so AND se AND sg = 0 [or not defined]) OR (us AND ue)
```

If either of these requirements is not fulfilled, the ABC editor will complain that your terminal is too dumb. One common problem on terminals with resizable windows is that the ABC prompt shows up like

```
>>>
  ?
```

on two lines instead of one. This means that the "li#" entry in your TERMCAP does not accurately reflect the number of lines actually in the window. This can be remedied by changing the setting of your TERMCAP environment variable, using the output of "stty size" (see stty(1)). (On systems that have the TIOCGWINSZ ioctl, we use it to get the proper window size; see tty(4) on BSD UNIX systems).

Error Messages

The error messages that ABC displays are all gathered in a file and only read when necessary. This was done to diminish the store used for all these strings and to enhance the adaptability of ABC to another natural language.

If you want the error messages in another language, for example French, you only have to replace the file `./abc.msg` by a French version. You might also change the manual in `./unix/abc.l` in that case; the helpfile `./abc.hlp` is created automatically from this manual.

NAME

abc — ABC interpreter & environment
abckeys — change key bindings for 'abc'

SYNOPSIS

abc [workspace and editor options] [file ...]
abc [workspace and task options]
abckeys

DESCRIPTION

Without options or files, the ABC interpreter is started, using the ABC editor, in the last workspace used or in workspace '*first*' if this is your first abc session. A workspace is kept as a group of files in a directory, with separate files for each how-to and location. The workspace directories themselves are kept by default in the directory \$HOME/abc. On non-Unix machines, \$HOME is the disk you are working on.

Workspace Options:

-W dir use group of workspaces in 'dir' instead of \$HOME/abc.
-w name start in workspace 'name' instead of last workspace used.
-w path use 'path' as workspace (no -W option allowed).

Editor option:

-e Use \$EDITOR as editor to edit definitions, instead of ABC editor (Unix only).
file ... Read commands from file(s) instead of from standard input; input for READ commands is taken from standard input. If a file is called '-' and standard input is the keyboard, the ABC system is started up interactively for that entry.

Special tasks:

-i tab Fill table 'tab' with text lines from standard input
-o tab Write text lines from table 'tab' to standard output
-l List the how-to's in workspace on standard output
-r Recover a workspace when its index is lost: useful after a machine crash if the ABC internal administration files didn't get written out.
-R Recover the index of a group of workspaces

USAGE

(This is necessarily a very brief description; see 'The ABC Programmer's Handbook' for full details.)

Use 'QUIT' to finish an ABC session.

When ABC starts up interactively, it displays a prompt and awaits input.

TYPING AND SUGGESTIONS: as you type, the system tries to suggest a possible continuation for what you have typed; to accept the suggestion, press [accept] (by default this is bound to the [TAB] key; type '?' to find out the bindings for the keyboard you are using). If you don't want to accept the suggestion, just carry on typing (you can always type character for character, ignoring the suggestions). Usually the system knows where a letter must be capital and where not, and you usually don't have to use the shift key; however, in the few places where both a lower-case and an upper-case letter would be legal (for instance for AND), you have to type the letter upper-case.

When you type a control command, like WHILE, the system provides indentation automatically for the body of the command; to reduce the indentation one level, type [return].

CORRECTING AND EDITING: the [undo] key (by default bound to backspace) undoes the last key you typed. Repeatedly typing it undoes more and more, up to a certain maximum number of keypresses.

To correct other parts, you must put the 'focus' onto the part you want to change. The focus is displayed by underlining or reverse video. [Widen] and [extend] make the focus larger, [first] and [last] make it smaller.

[Delete] deletes the contents of the focus.

[Copy] copies the contents of the focus to a buffer, or if the focus is not focussed on anything, copies the contents of the buffer back to where you are positioned.

MOVING THE FOCUS: [Upline] and [downline] focus on one line above or below. [Previous] and [next] move the focus left and right. [Up], [down], [left], and [right] move an empty focus around. [Goto] widens the focus to the largest thing at the current position.

OTHER OPERATIONS: [Look] redraws the screen; [record] records all keystrokes until the next time you press [record] — [play] replays them. [Redo] redoes the last key(s) undone; [interrupt] interrupts a running command.

WORKSPACES: To create a new workspace, or go to an existing workspace, type '>name'. To go to the last workspace you were in, type a single '>'. To get a list of workspace names, type '>>'.

HOW-TO'S: To create a new how-to, just type the first line of the how-to. This creates the new how-to, and allows you to type the body. Use [exit] to finish it (by default [ESC][ESC]).

To visit a how-to, type a colon, followed by the name of the how-to. Again, use [exit] to exit. To visit the last how-to again, or the last how-to you got an error message for, type a single ':'. To get a list of the how-to's in this workspace, type '::'.

To edit a location, type a '=' followed by the name of the location. To re-edit it, type a single '='. To get a list of the locations in the workspace, type '=:'.

KEY BINDINGS

The binding of editing operations like [accept] to keys may be different for your keyboard; type a '?' at the prompt to find out what the bindings are for your keyboard.

To redefine the keys used for editor operations, run 'abckey'. This produces a private key definitions file. You will be given instructions on how to use it.

Keys labeled *f1..f8* are *function keys*. On Unix, the way to type these is terminal-dependent. The codes they send must be defined by the termcap entry for your terminal.

If a terminal has arrow keys ↑, ←, →, ↓ which transmit codes to the computer, these should be used for Up, Down, Left and Right. Again, the termcap entry must define the codes.

The Goto operation is of most use if the cursor can be moved locally at the terminal, or if the terminal has a mouse; the Goto operation will sense the terminal for the cursor or mouse position. On Unix, we use two extra non-standard termcap capabilities for this: 'sp' which gives the string that must be sent to the terminal to sense the cursor position, and 'cp' which defines the format of the reply (in the same format as other cursor-addressing strings in termcap). If your terminal's mouse-click sends the position of the click automatically, just set 'sp' to the empty string. See *termcap(5)* for more details.

FILES

\$HOME/copybuf.abc	copy buffer between sessions
\$HOME/abc/wsgroup.abc	table mapping workspace names to directory names
\$HOME/abc/abckey_\$TERM	private key definitions file (Unix only)
\$HOME/abc/abc.key	private key definitions file (non-Unix)
position.abc	focus position of edited how-to's in workspace
perm.abc	table mapping object names to file names
suggest.abc	suggestion list for user-defined commands
types.abc	table with codes for typechecking between how-to's
*.cmd	command how-to's in this workspace
*.zfd, *.mfd, *.dfd	function how-to's in this workspace
*.zpd, *.mpd, *.dpd	predicate how-to's in this workspace
*.cts	permanent locations in this workspace

abc.msg messages file, used for errors (not on Macintosh)
abc.hlp helpfile with this text (MacABC.help on Macintosh)

The latter two are searched for first in your startup directory, then in \$HOME/abc, and finally, on Unix, in a directory determined by the installer of ABC. On the IBM PC and Atari ST the directories in your \$PATH are used in the last stage (if you have a hard disk place these files in the workspaces directory abc).

ATARI ST IMPLEMENTATION

There are four files supplied: the program abc.tos itself, abckeys.tos for changing your key bindings, the help file abc.hlp, and the error messages file abc.msg. (See FILES above.) If you start ABC up from the desktop, and you want to use the options given above, like -w, you should rename abc.tos to abc.ttp. There is an additional facility for redirecting input and output: the parameter >outfile redirects all output from ABC to the file called outfile, and similarly <infile takes its input from the file called infile.

IBM PC IMPLEMENTATION

There are four files for running ABC, the program abc.exe itself, abckeys.exe for changing your key bindings, the help file abc.hlp, and the error messages file abc.msg. (See FILES above.) If your screen size is non-standard, or your machine is not 100% BIOS compatible (which is unusual these days), you can specify the screen-size, and whether to use the BIOS or ANSI.SYS for output, by typing after the A> prompt, before you start ABC up, one of the following:

```
SET SCREEN=ANSI lines cols  
SET SCREEN=BIOS lines cols
```

If you are going to use ANSI.SYS, be sure you have the line
DEVICE=ANSI.SYS

in your CONFIG.SYS file. Consult the DOS manual for further details.

APPLE MACINTOSH IMPLEMENTATION

There are three files supplied: MacABC, the application itself, MacABC.help, the help file, and MacABC.doc, a MacWrite document containing a variant of this text. The help file should be in the same folder as MacABC, or in your System Folder.

MacABC runs in a single window. You'll notice that most operations are menu entries, as well as being possible from the keyboard. You can start ABC up by double-clicking the MacABC icon in which case you start up in the last workspace used, or by double-clicking on any icon in a workspace, in which case you start in that workspace. In this latter case, if the filename of the icon you clicked on ends in .cmd, that how-to is executed, but the how-to may not have any parameters.

Instead of the special option flags mentioned above, most of the tasks, like recovering a workspace, can be done from the File menu.

† Notes for Macintosh guru's:

The messages are STR# resources in MacABC; you must use a resource editor to change them.

MacABC uses Monaco 9 for the screen, and Courier 10 for printing. You can change them with ResEdit, by editing the resource with type Conf and ID 0. The horizontal and vertical window-size and the window-title can also be adapted there. To facilitate this, first Paste the TMPL resource with ID 5189 named Conf from MacABC to (a copy of) ResEdit. But beware, MacABC only works properly with Fixed-width Fonts like Monaco and Courier.

SEE ALSO

Leo Geurts, Lambert Meertens and Steven Pemberton, *The ABC Programmer's Handbook*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989, ISBN 0-13-000027-2.

Steven Pemberton, *An Alternative Simple Language and Environment for PCs*, IEEE Software, Vol. 4, No. 1, January 1987, pp. 56-64.

The ABC Newsletter. Available free from CWI.

AUTHORS

Frank van Dijk, Leo Geurts, Timo Krijnen, Lambert Meertens, Steven Pemberton, Guido van Rossum.

ADDRESS

ABC Distribution, CWI/AA, Postbox 4079, 1009 AB Amsterdam, The Netherlands.
E-mail: 'abc@cwi.nl'.

SUMMARY OF SPECIAL ACTIONS

<i>:name</i>	Visit how-to called 'name'
<i>:</i>	Visit last how-to referred to
<i>::</i>	Display headings of how-to's in this workspace
<i>=name</i>	Visit contents of location
<i>=</i>	Visit last location visited
<i>==</i>	Display names of permanent locations in this workspace
<i>>name</i>	Visit workspace 'name'
<i>></i>	Visit last workspace visited
<i>>></i>	Display list of workspace names
QUIT	Leave ABC

SUMMARY OF EDITING OPERATIONS

Name	Default Keys†	Short description
Accept	[TAB]	Accept suggestion, focus to hole or end of line
Return	[RETURN]	Add line or decrease indentation
Widen	<i>f1</i> , [ESC] w	Widen focus
Extend	<i>f2</i> , [ESC] e	Extend focus (usually to the right)
First	<i>f3</i> , [ESC] f	Move focus to first contained item
Last	<i>f4</i> , [ESC] l	Move focus to last contained item
Previous	<i>f5</i> , [ESC] p	Move focus to previous item
Next	<i>f6</i> , [ESC] n	Move focus to next item
Upline	<i>f7</i> , [ESC] u	Move focus to whole line above
Downline	<i>f8</i> , [ESC] d	Move focus to whole line below
Up	↑, [ESC] U	Make new hole, move up
Down	↓, [ESC] D	Make new hole, move down
Left	←, [ESC] ,	Make new hole, move left
Right	→, [ESC] .	Make new hole, move right
Goto	[ctrl-G], <i>mouseclick</i>	New focus at cursor position
Undo	[BACKSPACE]	Undo effect of last key pressed (may be repeated)
Redo	[ctrl-U]	Redo last UNDOne key (may be repeated)
Copy	<i>f9</i> , [ctrl-C], [ESC]c	Copy buffer to hole, or focus to buffer
Delete	[ctrl-D]	Delete contents of focus (to buffer if empty)
Record	[ctrl-R]	Start/stop recording keystrokes
Play	[ctrl-P]	Play back recorded keystrokes
Look	[ctrl-L]	Redisplay screen
Help	<i>f10</i> , [ESC]?	Print summary of editing operations
Exit	[ctrl-X]	Finish changes or execute command
Interrupt (as set by 'stty')		Interrupt command execution
Suspend (as set by 'stty')		Suspend ABC (only for shell with job control)

† Notes:

[Ctrl-D] means: hold the [CTRL] (or [CONTROL]) key down while pressing d.
[ESC] w means: press the [ESC] key first, then w.

ABC QUICK REFERENCE

COMMANDS

WRITE <i>expr</i>	Write to screen; / before or after <i>expr</i> gives new line
READ <i>address</i> EG <i>expr</i>	Read expression from terminal to <i>address</i> ; <i>expr</i> is example
READ <i>address</i> RAW	Read line of text
PUT <i>expr</i> IN <i>address</i>	Put value of <i>expr</i> in <i>address</i>
SET RANDOM <i>expr</i>	Start random sequence for random and choice
REMOVE <i>expr</i> FROM <i>list</i>	Remove one element from <i>list</i>
INSERT <i>expr</i> IN <i>list</i>	Insert in right place
DELETE <i>address</i>	Delete permanent location or table entry
PASS	Do nothing
KEYWORD <i>expr</i> KEYWORD ...	Execute user-defined command
KEYWORD	Execute refined command
CHECK <i>test</i>	Check <i>test</i> and stop if it fails
IF <i>test</i> :	If <i>test</i> succeeds, execute <i>commands</i> ;
<i>commands</i>	no ELSE allowed
SELECT :	Select one alternative:
<i>test</i> : <i>commands</i>	try each <i>test</i> in order
...	(one must succeed;
<i>test</i> : <i>commands</i>	the last <i>test</i> may be ELSE)
WHILE <i>test</i> :	As long as <i>test</i> succeeds
<i>commands</i>	execute <i>commands</i>
FOR <i>name</i> ,... IN <i>train</i> :	Take each element of <i>train</i> in turn
<i>commands</i>	

HOW-TO's

HOW TO KEYWORD ... :	Define new command KEYWORD ...
<i>commands</i>	
HOW TO RETURN <i>f</i> :	Define new function <i>f</i> with no arguments
<i>commands</i>	(returns a value)
HOW TO RETURN <i>f</i> <i>x</i> :	Define new function <i>f</i> with one argument
<i>commands</i>	
HOW TO RETURN <i>x</i> <i>f</i> <i>y</i> :	Define new function <i>f</i> with two arguments
<i>commands</i>	
HOW TO REPORT <i>pr</i> :	Define new predicate <i>pr</i> with no arguments
<i>commands</i>	(succeeds/fails)
HOW TO REPORT <i>pr</i> <i>x</i> :	Define new predicate <i>pr</i> with one argument
<i>commands</i>	
HOW TO REPORT <i>x</i> <i>pr</i> <i>y</i> :	Define new predicate <i>pr</i> with two arguments
<i>commands</i>	
SHARE <i>name</i> ,...	Share permanent locations (before commands of how-to)

Refinements (after the commands of a how-to)

KEYWORD : <i>commands</i>	Define command refinement
<i>name</i> : <i>commands</i>	Define expression- or test-refinement

Terminating commands

QUIT	Leave command how-to or command refinement, or leave ABC
RETURN <i>expr</i>	Leave function how-to or expression refinement, return value of <i>expr</i>

REPORT <i>test</i>	Leave predicate how-to or test-refinement, report outcome of <i>test</i>
SUCCEED	The same, report success
FAIL	The same, report failure
EXPRESSIONS AND ADDRESSES	
666, 3.14, 3.14e-9	Exact constants
<i>expr</i> , <i>expr</i> , ...	Compound
<i>name</i> , <i>name</i> , ...	Naming (may also be used as address)
<i>text</i> @ <i>p</i>	"ABCD"@2 = "BCD" (also address)
<i>text</i> <i>q</i>	"ABCD" 3 = "ABC" (also address)
<i>text</i> @ <i>p</i> <i>q</i>	"ABCD"@2 1 = "BCD" 1 = "B"
<i>table</i> [<i>expr</i>]	Table selection (also address)
"Jan", 'Feb', 'Won''t!'	Textual displays (empty: "" or '')
"value = `expr`;"	Conversion of <i>expr</i> to text
{1; 2; 2; ...}	List display (empty: {})
{1..9; ...}, {"a".."z"; ...}	List of consecutive values
{["Jan"]: 1; ["Feb"]: 2; ...}	Table display (empty: {})
<i>f</i> , <i>f</i> <i>x</i> , <i>x</i> <i>f</i> <i>y</i>	Result of function <i>f</i> (no permanent effects)
<i>name</i>	Result of refinement (no permanent effects)
TESTS	
<i>x</i> < <i>y</i> , <i>x</i> <= <i>y</i> , <i>x</i> >= <i>y</i> , <i>x</i> > <i>y</i>	Order tests
<i>x</i> = <i>y</i> , <i>x</i> <> <i>y</i>	(<> means 'not equals')
0 <= <i>d</i> < 10	
<i>pr</i> , <i>pr</i> <i>x</i> , <i>x</i> <i>pr</i> <i>y</i>	Outcome of predicate <i>pr</i> (no permanent effects)
<i>name</i>	Outcome of refinement (no permanent effects)
<i>test</i> AND <i>test</i> AND ...	Fails as soon as one of the tests fails
<i>test</i> OR <i>test</i> OR ...	Succeeds as soon as one of the tests succeeds
NOT <i>test</i>	
SOME <i>name</i> , ... IN <i>train</i> HAS <i>test</i>	Sets <i>name</i> , ... on success
EACH <i>name</i> , ... IN <i>train</i> HAS <i>test</i>	Sets <i>name</i> , ... on failure
NO <i>name</i> , ... IN <i>train</i> HAS <i>test</i>	Sets <i>name</i> , ... on failure

PREDEFINED FUNCTIONS AND PREDICATES

Functions and predicates on numbers

~ <i>x</i>	Approximate value of <i>x</i>
exactly <i>x</i>	Exact value of <i>x</i>
exact <i>x</i>	Test if <i>x</i> is exact
+ <i>x</i> , <i>x</i> + <i>y</i> , <i>x</i> - <i>y</i> , - <i>x</i> , <i>x</i> * <i>y</i> , <i>x</i> / <i>y</i>	Plain arithmetic
<i>x</i> ** <i>y</i>	<i>x</i> raised to the power <i>y</i>
root <i>x</i> , <i>n</i> root <i>x</i>	Square root, <i>n</i> -th root
abs <i>x</i> , sign <i>x</i>	Absolute value, sign (= -1, 0, or +1)
round <i>x</i> , floor <i>x</i> , ceiling <i>x</i>	Rounded to whole number
<i>n</i> round <i>x</i>	<i>x</i> rounded to <i>n</i> digits after decimal point
<i>a</i> mod <i>n</i>	Remainder of <i>a</i> on division by <i>n</i>
*/ <i>x</i>	Numerator of exact number <i>x</i>
/ <i>**x</i>	Denominator
random	Random approximate number <i>r</i> , 0 ≤ <i>r</i> < 1

e, *exp x*
log x, *b log x*
pi, *sin x*, *cos x*, *tan x*, *arctan x*
angle (x, y), *radius (x, y)*
c sin x, *c cos x*, *c tan x*
c arctan x, *c angle (x, y)*
now

Functions on texts

t^u
t^n
lower t
upper t
stripped t
split t

Function on tables

keys table

Functions and predicates on trains

#train
e#train
e in train, *e not in train*
min train
e min train
max train, *e max train*
train item n
choice train

Functions on all types

x<<n
x><n
x>>n

THE CHARACTERS

!"#\$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~

Base of natural logarithm, exponential function
Natural logarithm, logarithm to the base *b*
Trigonometric functions, with *x* in radians
Angle of and radius to point (*x, y*)
Similar, with the circle divided into *c* parts
(e.g. 360 for degrees)
e.g. (1999, 12, 31, 23, 59, 59.999)

t and *u* joined into one text
t repeated *n* times
lower "aBc" = "abc"
upper "aBc" = "ABC"
Strip leading and trailing spaces from *t*
Split text *t* into words

List of all keys in *table*

Number of elements in *train*
Number of elements equal to *e*
Test for presence or absence
Smallest element of *train*
Smallest element larger than *e*
Largest element
n-th element
Random element

x converted to text, aligned left in width *n*
The same, centred
The same, aligned right

This is the order of all characters
that may occur in a text.
(The first is a space.)

A Short Introduction to the ABC Language

While the full documentation about ABC is in the ABC Programmer's Handbook, this article should give you just enough information to get going.

THE LANGUAGE

ABC is an imperative language originally designed as a replacement for BASIC: interactive, very easy to learn, but structured, high-level, and easy to use. ABC has been designed iteratively, and the present version is the 4th iteration. The previous versions were called *B* (not to be confused with the predecessor of C).

It is suitable for general everyday programming, the sort of programming that you would use BASIC, Pascal, or AWK for. It is not a systems-programming language. It is an excellent teaching language, and because it is interactive, excellent for prototyping. It is much faster than 'bc' for doing quick calculations.

ABC programs are typically very compact, around a quarter to a fifth the size of the equivalent Pascal or C program. However, this is not at the cost of readability, on the contrary in fact (see the examples below).

ABC is simple to learn due to the small number of types in the language (five). If you already know Pascal or something similar you can learn the whole language in an hour or so. It is easy to use because the data-types are very high-level.

The five types are:

- numbers: unbounded length, with exact arithmetic the rule
- texts (strings): also unbounded length
- compounds: records without field names
- lists: sorted collections of any one type of items (bags or multi-sets)
- tables: generalised arrays with any one type of keys, any one type of items (finite mappings).

THE ENVIRONMENT

The implementation includes a programming environment that makes producing programs very much easier, since it knows a lot about the language, and can therefore do much of the work for you. For instance, if you type a W, the system suggests a command completion for you:

```
W?RITE ?
```

If that is what you want, you press [tab], and carry on typing the expression; if you wanted WHILE, you type an H, and the system changes the suggestion to match:

```
WH?ILE ?:
```

This mechanism works for commands you define yourself too. Similarly, if you type an open bracket or quote, you get the closing bracket or quote for free. You can ignore the suggestions if you want, and just type the commands full out.

There is support for workspaces for developing different programs. Within each workspace variables are persistent, so that if you stop using ABC and come back later, your variables are still there as you left them. This obviates the need for file-handling facilities: there is no conceptual difference between a variable and a file in ABC.

The language is strongly-typed, but without declarations. Types are determined from context.

EXAMPLES

The (second) best way to appreciate the power of ABC is to see some examples (the first is to use it). In what follows, >>> is the prompt from ABC.

NUMBERS

```
>>> WRITE 2**1000
107150860718626732094842504906000181056140481170553360744375038837
035105112493612249319837881569585812759467291755314682518714528569
231404359845775746985748039345677748242309854210746050623711418779
541821530464749835819412673987675591655439460770629145711964776865
42167660429831652624386837205668069376

>>> PUT 1/(2**1000) IN x
>>> WRITE 1 + 1/x
107150860718626732094842504906000181056140481170553360744375038837
035105112493612249319837881569585812759467291755314682518714528569
231404359845775746985748039345677748242309854210746050623711418779
541821530464749835819412673987675591655439460770629145711964776865
42167660429831652624386837205668069377
```

TEXTS

```
>>> PUT ("ha " ^^ 3) ^ ("ho " ^^ 3) IN laugh
>>> WRITE laugh
ha ha ha ho ho ho

>>> WRITE #laugh
18

>>> PUT "Hello! "^^1000 IN greeting
>>> WRITE #greeting
7000
```

LISTS

```
>>> WRITE {1..10}
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
>>> PUT {1..10} IN l
>>> REMOVE 5 FROM l
>>> INSERT pi IN l
>>> WRITE l
{1; 2; 3; 3.141592653589793; 4; 6; 7; 8; 9; 10}

>>> PUT {} IN ll
>>> FOR i IN {1..3}:
    INSERT {1..i} IN ll
>>> WRITE ll
{{1}; {1; 2}; {1; 2; 3}}
>>> FOR l IN ll:
    WRITE l /

{1}
{1; 2}
{1; 2; 3}
>>> WRITE #ll
3
```


COMPOUNDS

```
>>> PUT ("Square root of 2", root 2) IN c
>>> WRITE c
("Square root of 2", 1.414213562373095)
>>> PUT c IN name, value
>>> WRITE name
Square root of 2
>>> WRITE value
1.414213562373095
```

A TELEPHONE LIST

This uses the table data-type. In use, tables resemble arrays:

```
>>> PUT {} IN tel
>>> PUT 4054 IN tel["Jennifer"]
>>> PUT 4098 IN tel["Timo"]
>>> PUT 4134 IN tel["Guido"]

>>> WRITE tel["Jennifer"]
4054
```

You can write all ABC values out. Tables are kept sorted on the keys:

```
>>> WRITE tel
{"Guido": 4134; ["Jennifer": 4054; ["Timo": 4098}
```

The "keys" function returns a list:

```
>>> WRITE keys tel
{"Guido"; "Jennifer"; "Timo"}

>>> FOR name IN keys tel:
    WRITE name, ":", tel[name] /
Guido: 4134
Jennifer: 4054
Timo: 4098
```

You can define your own commands:

```
HOW TO DISPLAY t:
    FOR name IN keys tel:
        WRITE name<<10, tel[name] /

>>> DISPLAY tel
Guido      4134
Jennifer   4054
Timo       4098
```

To find the user of a given number, you can use a quantifier:

```
>>> IF SOME name IN keys tel HAS tel[name] = 4054:
    WRITE name
Jennifer
```

Or create the inverse table:

```
>>> PUT {} IN subscriber
>>> FOR name IN keys tel:
    PUT name IN subscriber[tel[name]]

>>> WRITE subscriber[4054]
Jennifer
>>> WRITE subscriber
{[4054]: "Jennifer"; [4098]: "Timo"; [4134]: "Guido"}
```

Commands and functions are polymorphic:

```
>>> DISPLAY subscriber
4054      Jennifer
4098      Timo
4134      Guido
```

Functions may return any type. Note that indentation is significant - there are no BEGIN-END's or { }'s:

```
HOW TO RETURN inverse t:
  PUT {} IN inv
  FOR k IN keys t:
    PUT k IN inv[t[k]]
  RETURN inv

>>> WRITE inverse tel
{[4054]: "Jennifer"; [4098]: "Timo"; [4134]: "Guido"}
>>> DISPLAY inverse inverse tel
Guido      4134
Jennifer   4054
Timo       4098
```

A CROSS-REFERENCE INDEXER

'Text files' are represented as tables of numbers to strings:

```
>>> DISPLAY poem
1      I've never seen a purple cow
2      I hope I never see one
3      But I can tell you anyhow
4      I'd rather see than be one
```

The following function takes such a document, and returns the cross-reference index of the document: a table from words to lists of line-numbers:

```
HOW TO RETURN index doc:
  PUT {} IN where
  FOR line.no IN keys doc:
    TREAT LINE
  RETURN where
TREAT LINE:
  FOR word IN split doc[line.no]:
    IF word not.in keys where:
      PUT {} IN where[word]
    INSERT line.no IN where[word]
```

TREAT LINE here is a refinement, directly supporting stepwise-refinement. "split" is a function that splits a string into its space-separated words:

```
>>> WRITE split "Hello world"  
{[1]: "Hello"; [2]: "world"}
```

```
>>> DISPLAY index poem
```

```
But      {3}  
I        {2; 2; 3}  
I'd      {4}  
I've     {1}  
a        {1}  
anyhow   {3}  
be       {4}  
can      {3}  
cow      {1}  
hope     {2}  
never    {1; 2}  
one      {2; 4}  
purple   {1}  
rather   {4}  
see      {2; 4}  
seen     {1}  
tell     {3}  
than     {4}  
you      {3}
```

An Adventure Program

Steven Pemberton
CWI, Amsterdam

Adventure style games are very popular in computing circles, and I'm going to develop a small one here. Because of space I will have to leave out many of the advanced features of most adventure games, but it will give you an idea of how it looks in ABC. And of course it will be obvious how the bells and whistles can quickly be added on.

As I'm sure you know, a (textual) adventure program works by describing a scene. You then give instructions on where to go, or what to do, and it responds by telling you what happened as a result. For instance, if it says

*You are standing by a building at the end of a road.
A spring flows from the building.*

and you reply

enter building

it might reply

*You are inside a building, a well house for a spring.
There is a bottle here.
There are some keys here.*

after which the dialogue might proceed as follows:

> take keys
> leave the building
Please use 1 or 2 word sentences.
> leave
You are outside the building.
> go west
You are standing by a stream.
> go south
*You are at a small slit that the stream runs down.
A dry river bed carries on ahead.*
> go down
You don't seem to be able to go that way.
> south
You have found a metal grate fixed into the ground.
> down
Sorry, you can't do that.
> open grate
*The grate is open.
You are at a hole in the ground.
There is a metal grate lying on the ground.*
> down
*You are in a dim chamber.
A hole in the ceiling shows the sky above.*

and so on.

The main program making up this adventure looks like this:

```
HOW TO ADVENTURE:
  START
  GET command
  WHILE command <> "quit":
    OBEY command
    GET command
  FINISH
```

START will initialise some variables, like the place where the player is, and what the player is holding. FINISH will print out the score and so on. GET will print the prompt, read a line, strip off spaces, and reduce it to lower-case:

```
HOW TO GET command:
  GET LINE
  WHILE command = "": GET LINE
GET LINE:
  WRITE "> "
  READ command RAW
  PUT lower stripped command IN command
```

OBEY has to split a command into its constituent words, and then decide what action needs to be taken for that command:

```
HOW TO OBEY command:
  SPLIT command INTO verb AND object
  SELECT:
    verb = "": PASS
    special command: TRY TO MOVE command
    verb = "move": TRY TO MOVE object
    verb = "take": TRY TO TAKE object
    verb = "drop": TRY TO DROP object
    verb = "kill": TRY TO KILL object
    verb = "what": INVENTORY
  ELSE: CAN'T DO verb, object
```

SPLIT does what its name suggests: splits the command into its constituent words, and makes sure it only consists of one or two words:

```
HOW TO SPLIT command INTO verb AND object:
  PUT split command IN words
  SELECT:
    #words = 1: PUT words item 1, "" IN verb, object
    #words = 2: PUT words item 1, words item 2 IN verb, object
  ELSE:
    WRITE "Please use 1 or 2 word sentences." /
    PUT "", "" IN verb, object
```

A nice feature is to allow synonyms for commands, to allow "go west" and "proceed west" and "move west" all to mean the same thing. We can do that by having a table of synonyms:

```
>>> WRITE synonyms["move"]
{"go"; "proceed"}
```

and then adding in SPLIT:

SHARE synonyms

```
...  
IF SOME word IN keys synonyms HAS verb in synonyms[word]:  
    PUT word IN verb
```

Moving places

In this adventure, each place has a name, which is a short description you get each time you visit it after the first time ("You are x", such as "outside the building" above). Then, each place has a long description used for describing it the first time you go there. Such a description is stored as a table of lines, for instance

```
>>> WRITE description["inside large hall"]  
    {[1]: "This is a large hall."; [2]: "There is an exit to the west."}
```

To display such a message neatly, we can define the following command:

```
HOW TO DISPLAY message:  
    FOR line IN message:  
        WRITE line /
```

```
>>> DISPLAY description["inside large hall"]  
    This is a large hall.  
    There is an exit to the west.
```

Then there is a map of all locations, which gives for each location a table of directions that the player can go in, and where that direction leads to.

```
>>> WRITE map["inside the building"]  
    {"out": "outside the building"}  
>>> WRITE map["outside the building"]  
    {"in": "inside the building"; ["south": "standing by the stream";  
    ["west": "in the forest"]}
```

We can play a nasty trick on the player:

```
>>> WRITE map["in the forest"]  
    {"east": "in the forest"; ["north": "in the forest";  
    ["south": "in the forest"; ["west": "standing by the stream"]}
```

Moving is attempted by means of the command TRY TO MOVE. All commands beginning TRY TO first check that the conditions for the action are acceptable, and only then do the action. The current location is held in place: TRY TO MOVE checks that the direction asked for is in the map for the current place:

```
HOW TO TRY TO MOVE direction:  
    SHARE map, place  
    SELECT:  
        direction = "":  
            WRITE "Where to?" /  
        direction in keys map[place]:  
            MOVE TO map[place][direction]  
    ELSE:  
        WRITE "You don't seem to be able to go that way" /
```

MOVE TO does the actual moving. For now here is a simple version, but it will get more involved later.

```
HOW TO MOVE TO there:  
    SHARE place  
    PUT there IN place  
    DESCRIBE place
```

(DESCRIBE describes a place and the objects to be found there; you'll see it shortly.)

In OBEY, you will have noticed the lines

```
SELECT:
    special command: TRY TO MOVE command
```

This is to allow the player to say `south` instead of `go south`, by seeing if the command is already in the map for the current place:

```
HOW TO REPORT special command:
    SHARE map, place
    REPORT command in keys map[place]
```

Notice that it also allows you to use commands instead of directions in the map. For instance, when at the grate, you can open the grate by having two places, an open grate and a closed grate:

```
>>> WRITE map["at closed grate"]
    [{"north": "at slit"; ["open grate": "at open grate"}]
```

Taking and dropping objects

Different objects are left lying about at various places. These are recorded in a table `objects`. Just as with places, each object has a simple name, to be used when the player wants to know what is being carried, and a longer description when an object is first found.

```
>>> WRITE objects["inside the building"]
    {"bottle"; "keys"}
>>> WRITE description["keys"]
    There are some keys here.
```

Now I can show you `DESCRIBE`. It remembers which places have already been described (and therefore visited), and only gives the long description the first time:

```
HOW TO DESCRIBE place:
    SHARE description, objects, visited
    SELECT:
        place in visited:
            WRITE "You are ", place /
        ELSE:
            DISPLAY description[place]
            INSERT place IN visited
    FOR object IN objects for place:
        DISPLAY description[object]
```

Notice here the line `"FOR object IN objects for place:"`. Not every place may be recorded in the `objects` table, so it is a shorthand to save repeated checks to see if it is:

```
HOW TO RETURN property for thing:
    SELECT:
        thing in keys property: RETURN property[thing]
        ELSE: RETURN {}
```

You'll find it used again later on.

Then there is a list of what the player is carrying, called `holding`, which is initially empty. To find out what is being carried, the player can ask for an inventory:

HOW TO INVENTORY:

```
SHARE holding
SELECT:
  holding = {}:
    WRITE "You aren't carrying anything" /
  ELSE:
    WRITE "You are carrying: "
    LIST holding
```

This uses a useful command to neatly print a list of objects:

HOW TO LIST things:

```
PUT "" IN separator
FOR object IN things:
  WRITE separator, object
  PUT ", " IN separator
WRITE /
```

```
>>> LIST objects["inside the building"]
bottle, keys
```

Another useful tool is a test to see if an object is currently being carried:

```
HOW TO REPORT carrying object:
SHARE holding
REPORT object in holding
```

and another to test if an object is present:

```
HOW TO REPORT present object:
SHARE objects, place
REPORT object in objects for place
```

TRY TO TAKE can now check that the object is present, that it's not already being carried and so on, before actually taking it:

```
HOW TO TRY TO TAKE object:
SHARE holding
SELECT:
  object = "":
    WRITE "Which object?" /
  carrying object:
    WRITE "You're already carrying it!" /
  NOT present object:
    WRITE "I see no `object`." /
  #holding > 6:
    WRITE "You can't carry any more." /
  ELSE:
    TAKE object
```

TAKE looks like this, again a simple version for now:

```
HOW TO TAKE object:
SHARE holding, objects, place
REMOVE object FROM objects[place]
INSERT object IN holding
```

TRY TO DROP is similar:

HOW TO TRY TO DROP object:

```
SELECT:
  object = "": WRITE "Which object?" /
  NOT carrying object: WRITE "You're not holding it!" /
  ELSE: DROP object
```

HOW TO DROP object:

```
SHARE holding, objects, place
REMOVE object FROM holding
INCLUDE object IN objects FOR place
```

The command INCLUDE adds an item to a table:

HOW TO INCLUDE object IN property FOR thing:

```
IF thing not.in keys property:
  PUT {} IN property[thing]
INSERT object IN property[thing]
```

Conditions and side effects

One of the tricks of adventure games is that certain actions are not possible unless you are at a certain place, or you are carrying a certain thing, and some actions have unexpected side-effects.

For instance, you shouldn't be able to open the grate if you aren't carrying the keys. So we can alter MOVE TO to check for this:

HOW TO MOVE TO there:

```
SHARE place
SELECT:
  opening.grate AND NOT carrying "keys":
    WRITE "I don't seem able to open the grate" /
  ELSE:
    PUT there IN place
    DESCRIBE place
```

opening.grate:

```
REPORT (place, there) = ("at closed grate", "at open grate")
```

Similarly, somewhere in the cave there is a bird, but you can only catch it if you're carrying the cage. Furthermore, the jangling of the keys frightens it. So we can alter TAKE to do this:

HOW TO TAKE object:

```
SHARE holding, objects, place
SELECT:
  object = "bird" AND carrying "keys":
    WRITE "The bird flutters off in fright." /
  object = "bird" AND NOT carrying "cage":
    WRITE "You don't seem able to catch the bird." /
  ELSE:
    REMOVE object FROM objects[place]
    INSERT object IN holding
```

An example of a side-effect is that dropping the bird is the only way to scare off the snake (should you meet it):

```
HOW TO DROP object:
SHARE holding, objects, place
IF object = "bird" AND present "snake":
    WRITE "With a great flurry the bird attacks the snake." /
    WRITE "The snake flees into the darkness." /
    REMOVE "snake" FROM objects[place]
REMOVE object FROM holding
INCLUDE object IN objects FOR place
```

(Obviously, TAKE should also be changed to prevent you from trying to take the snake.)

Removing objects with extreme prejudice

Now you've seen that there are living creatures in the cave. Certain of them are undesirable to the player's well-being and score, and in the brutal tradition of adventure games must be eliminated. Of course some are harmless, but computers only do what they are told...

```
HOW TO TRY TO KILL object:
SELECT:
    object = "":
        WRITE "Which object?" /
        (NOT present object) AND (NOT carrying object):
            WRITE "I see no `object`" /
    ELSE:
        KILL object
```

```
HOW TO KILL object:
SHARE holding, objects, place
SELECT:
    object = "bird":
        WRITE "How cruel! The poor bird dies with a mournful peep." /
        ELIMINATE
        INCLUDE "dead bird" IN objects FOR place
    object = "snake":
        WRITE "Attacking the snake is both dangerous and ineffective." /
    ELSE: \ It's not a living creature
        WRITE "It's already dead!" /
ELIMINATE:
    SELECT:
        carrying object: REMOVE object FROM holding
        present object: REMOVE object FROM objects[place]
```

Odds and ends

Well, that's the body of the adventure. Of course, lots of extra places, objects, beings and commands must be added, but that's just a case of more of the same.

In OBEY, if it can't obey your command, it invokes CAN'T DO. As a nicety this prints funny remarks for certain commands. For instance, if you're at the stream, you might try "swim":

```
>>> DISPLAY funny["swim"]
The water would get into my circuits.
```


HOW TO CAN'T DO verb:

SHARE funny

SELECT:

verb in keys funny: DISPLAY funny[verb]

ELSE:

WRITE "Sorry, you can't do that" /

As a final touch, you might want to add the commands "save" and "restore" to OBEY, so you can save a game, and come back later to it (or so you can try something, and if it fails restore it and try something else). This is remarkably easy. Since the state of the game is reflected by a small number of variables, you can just put them in another variable:

HOW TO SAVE:

SHARE saved, holding, objects, place

PUT holding, objects, place IN saved

HOW TO RESTORE:

SHARE saved, holding, objects, place

PUT saved IN holding, objects, place

DESCRIBE place

A Histogram Program

Jurjen Bos
CWI, Amsterdam

Here is a small example program. It shows a histogram on your screen. A call like

```
HISTOGRAM {"*"}: {"a": 5; "b": 6; "c": 3}; {"+": {"b": 5; "c": 1}}
```

will produce a view like:

```
6|           *
5|      *      * +
4|      *      * +
3|      *      * +           *
2|      *      * +           *
1|      *      * +           * +
-|-----
|      a           b           c
```

The program automatically scales the rows vertically and adjusts column layout. It uses some interesting datastructures; see for example the technique used to print a line in the refinement DRAW GRAPH.

```
HOW TO HISTOGRAM data: \Jurjen Bos, 891029
```

```
  PUT 24, 79 IN height, width \set to the size of your screen
```

```
  ANALYSE DATA
```

```
  SCALE
```

```
  COMPUTE COLUMN LAYOUT
```

```
  INDEX TABLES
```

```
  DRAW GRAPH
```

```
  BOTTOM LINES
```

```
ANALYSE DATA:
```

```
  \collect total length of symbols, column names, and graph data
```

```
  PUT 0, {}, {} IN symbols, columns, graph
```

```
  FOR symbol IN keys data:
```

```
    FOR col IN keys data[symbol]:
```

```
      IF col not.in columns: INSERT col IN columns
```

```
      INSERT data[symbol][col], symbol, col IN graph
```

```
    PUT symbols+#symbol IN symbols
```

```
SCALE:
```

```
  PUT max graph IN level, symbol, col
```

```
  PUT level/(height-3.5) IN scale
```

```
  PUT 10**max{0; floor(10 log scale)} IN ten.power
```

```
  IF (scale/ten.power) not.in round.numbers:
```

```
    PUT ((scale/ten.power) min round.numbers)*ten.power IN scale
```

```
  IF SOME i IN {0..2} HAS i round scale=scale: PUT i IN digits
```

```
round.numbers: RETURN {1; 1.25; 1.5; 2; 2.5; 3; 4; 5; 6; 8; 10}
```

```
COMPUTE COLUMN LAYOUT:
```

```
  PUT #``digits round level`` IN left
```

```
  PUT floor((width-left-1)/#columns) IN col.width
```

```
  SELECT:
```

```
    col.width>symbols+2*#data:
```

```
      PUT 2, min{col.width; symbols+2*#data+7} IN bar.space, col.width
```

```
    col.width>symbols+#data:
```

```
      PUT 1, min{col.width; symbols+#data+5} IN bar.space, col.width
```

```

col.width>=symbols:
  PUT 0, min{col.width; symbols+3} IN bar.space, col.width
ELSE:
  WRITE "The columns do not fit on the page. Sorry."
  CHECK 0=1
  PUT 1+floor((col.width-symbols-bar.space*#data+bar.space)/2) IN right
INDEX TABLES:
  \make table for column positions
  PUT {} IN col.index
  FOR col IN columns:
    PUT #col.index*col.width+right IN col.index[col]
  DELETE columns
  \compute table for bar positions in a column
  PUT 0, {} IN symbols, symbol.index
  FOR symbol IN keys data:
    PUT symbols IN symbol.index[symbol]
    PUT symbols+bar.space+#symbol IN symbols
DRAW GRAPH:
  PUT max graph IN level, symbol, col
  INSERT (0, "", col) IN graph
  PUT " ^^#col.index*col.width IN line
  PUT scale*round(level/scale) IN line.level
  WHILE line.level>0:
    WRITE digits round line.level>>left, "|"
    WHILE level>=line.level-scale/2:
      PUT symbol IN line@col.index[col]+symbol.index[symbol]|#symbol
      PUT (level, symbol, col)max graph IN level, symbol, col
    WRITE line /
    PUT line.level-scale IN line.level
BOTTOM LINES:
  WRITE "-^^left, "|", "-^^#line /
  WRITE " ^^left, "|"
  FOR col IN keys col.index: WRITE (col><col.width)|col.width
  WRITE /

```

I used the program to demonstrate to people how much the sum of a set of dice resembles the normal distribution. For this I used the following program. It will draw a histogram comparing the normal distribution, the real distribution of the dice, and a sample of 100 throws, all scaled to fit in the same diagram. The program allows you to throw "dice" with any number of sides: the sides will always have values from one onwards. Thus, a coin is a two-sided die with sides 1 and 2.

Here is the program:

```

HOW TO PICTURE n DICE OF sides SIDES: \ Dice thrower, Jurjen 891030
\first, compute distribution of ideal dice
PUT {[0]: 1} IN die
FOR counter IN {1..n}:
  PUT {} IN new
  FOR j IN keys die: PUT die[j] IN new[j+1]
  FOR i IN {2..sides}:
    PUT die[max keys die] IN new[(max keys die)+i]
    FOR j IN {min keys die .. (max keys die)-1}:
      PUT new[i+j]+die[j] IN new[i+j]
  PUT new IN die
  DELETE new
\compute normal approximation
PUT sides**n IN s

```

```

PUT n*(sides+1)/2, n*(sides**2-1)/12 IN mu, sigma
PUT {} IN norm
FOR i IN {n..n*sides}:
  PUT s*normal(mu, sigma, i) IN norm[i]
\compute sample
PUT {} IN sample
FOR i IN {n..n*sides}: PUT 0 IN sample[i]
FOR i IN {1..100}:
  PUT 0 IN x
  FOR counter IN {1..n}:
    PUT x+choice{1..sides} IN x
  PUT sample[x]+s/100 IN sample[x]
\off we go
WRITE "*: real dice +: normal simulation s: sample of 100"/
HISTOGRAM [{"*"}: die; [{"+"}: norm; [{"s"}: sample}

```

```

HOW TO RETURN normal(mu, sigma, x):
  RETURN (exp(-0.5*(x-mu)**2/sigma))/root(2*pi*sigma)

```

Now, imagine yourself throwing dice for the royal game of goose:

```

>>> PICTURE 2 DICE OF 6 SIDES
*: real dice +: normal simulation s: sample of 100
6|
5|
4|
3|
2|
1|
-----
| 2 3 4 5 6 7 8 9 10 11 12

```

Finally, a less trivial example:

```

>>> PICTURE 7 DICE OF 3 SIDES
*: real dice +: normal simulation s: sample of 100
510|
480|
450|
420|
390|
360|
330|
300|
270|
240|
210|
180|
150|
120|
90|
60|
30|
-----
| 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

```

How To Order ABC, Release 1

If you can't find ABC on any bulletin board or archive machine that you have access to, you can order a copy using this form, from:

ABC Implementations
CWI/AA
Postbox 4079
1009 AB Amsterdam
The Netherlands

ORDER FORM

Name:

Address:

.....

.....

Country:

For personal computers

Please send me a copy of the ABC implementation for:

- Apple Macintosh, 3½" single sided floppy disk (400K).
- Atari ST, 3½" single sided floppy disk (360K).
- IBM PC and compatibles running MS DOS, 5¼" floppy disk (360K).
- IBM PC and compatibles running MS DOS, 3½" floppy disk (720K).

For Unix

Please send me a copy of the Unix sources for the ABC implementation in tar format on:

- 600', ½" reel tape, 1600 bpi.
- ¼" cartridge tape, QIC-24 format.
- ¼" cartridge tape, QIC-11 format.

(For other media and formats, please inquire.)

Prices

Prices are in Dutch guilders for Europe, and US dollars for the rest of the world. They include all taxes and postage costs, and are correct at the time of printing:

- On floppy disk: fl. 35 within Europe \$25 elsewhere
- On reel tape: fl. 100 within Europe \$60 elsewhere
- On cartridge tape: fl. 250 within Europe \$150 elsewhere

I enclose a cheque or international money order for made payable to "Stichting Mathematisch Centrum, Amsterdam".

Signature and Date:

