Elizabeth 2.20

Chatbot Creation System

Professor Peter Millican

Hertford College, University of Oxford

ELIZABETH is an adaptation of Joseph Weizenbaum's famous 1966 ELIZA program, in which the various selection, substitution, and phrase storage mechanisms have been considerably enhanced and generalised to increase both its flexibility and its potential adaptability. The system also incorporates analysis tables to show exactly what processing has taken place, thus providing a learning tool that can give insights into some of the fundamental methods and issues of artificial intelligence within an entertaining context.

All comments, error reports, and suggestions for improvement in any respect will be gratefully received. Please send them to ***peter.millican@hertford.ox.ac.uk***

Table of Contents

1. Introduction to the *Elizabeth* System

1.1 Overview of the System

Elizabeth starts the conversation by choosing one of its predefined 'welcome' messages. From then on, its responses are generated from the user's inputs, by a process that typically involves the following steps:

(a) Systematic transformation of the input (by means of 'input transformations').

(b) Identification of 'key words' within the input, and choice of appropriate messages where this identification succeeds (so-called 'keyword transformations').

(c) Systematic transformation of the output (by means of 'output transformations' and 'final transformations' – these two types of transformation behave identically unless your script involves Recursion and Text Splitting, see §3.2).

All of these processes are governed by a 'Script', which is read from a text file in advance (typically within the 'My Scripts' subdirectory which you should use for your own work). The system comes with a default Script – in the file `Elizabeth.txt` – which is automatically read when the program starts (and if `Elizabeth.txt` is ever deleted, it is recreated from a copy named `EOriginal.txt` in the 'Illustrative Scripts' subdirectory). But alternative scripts can be loaded using the File menu, and you will learn most from the system if you design and experiment with your own, either extending the default Script or the others provided with the system, or (better still) creating your own from scratch. The best way of finding out about all this is probably to study the default script file `Elizabeth.txt` within the resident Script Editor (see §1.3) while referring to the 'Introduction to Script Commands' (§2.1) and the other sections referenced there. An annotated version of this default script, together with an example of the results it can produce, is provided as an Illustrative Script and Conversation (see §1.4 – if you select 'First illustrative script' from *Elizabeth's* Help menu, this will load the default Script). You may find it helpful to focus your initial exploration of the system by working through the 'Self-Teach Exercises' provided in Section §1.6.

You can see details of *Elizabeth's* internal processing by clicking on the 'Trace' tab in the *Elizabeth* display. If the 'Trace' tab is not visible, make it appear by selecting 'View analysis tables' from the Analysis menu, and note that the Analysis menu also enables you to include details of the operation of the matching algorithm (either during input, keyword, output, or final transformations) within the trace display.

For practical details of running the system and the various operations provided by the menus, see §1.2, entitled 'Configuration, Starting Up, and the System Menus'.

1.2 Configuration, Starting Up, and the System Menus

If you have unpacked the system from a zip file on the web (at http://philocomp.net/ai/elizabeth.htm), then it should already be configured appropriately, with directories and files in the default setup. Most of this document accordingly presupposes that default setup. However, if you have special requirements, for example needing to separate 'read only areas' of a network (where the software is provided) from 'user areas' (where users can edit their own files), please see the configuration details in §6.4 below.

When the system starts up, it automatically looks for the file `Elizabeth.txt` within the 'My Scripts' subdirectory and loads the Script that it contains. If no such file exists, then the system instead looks for the file `EOriginal.txt` in the 'Illustrative Scripts' subdirectory, which is intended to provide a backup copy of the default Script, and it then re-creates `Elizabeth.txt` from it before loading. (The advantage of this arrangement is that you can edit `Elizabeth.txt` to make the system start up with your own preferred Script, without losing the original version; if at any point you then delete your own version, the original will automatically return).

For initial familiarisation with the system, see the 'Overview of the System' in §1.1 and the sections referenced there (in particular the annotated version of `Elizabeth.txt` provided in the 'Illustrative Script and Conversation', §1.4). Below are explained the facilities available from the system menus.

1.2.1 The File Menu

Once the system is up and running, the File menu gives you the following options:

File | Load script file and start
Brings up a 'load file' selection dialog box, inviting you to choose a script file to be loaded (usually with the filename extension `txt`). Assuming you do choose such a file, the system will be completely re-initialised and the chosen Script read in before beginning a new conversation.

File | Restart with current script
Re-initialises the system and begins a new conversation with the currently loaded script file. **Ctrl-R** provides a keyboard shortcut for this.

File | Transfer script into Script Editor
Loads the current script file into the resident Script Editor (see §1.3) and transfers control to it. Although you don't have to use this resident editor for editing script files (*Notepad* or any other plain text editor can also be used), it has various advantages explained in §1.3. This command has a keyboard shortcut of **Ctrl-T** (and note that this command can be used to 'toggle' both ways between *Elizabeth* and the Script Editor, copying the contents from one to the other).

File | Switch into Script Editor
Switches control to the resident Script Editor (see §1.3), but without affecting what the Editor contains (so if you do this just after starting *Elizabeth*, the Editor will be empty). Note that the keyboard shortcut **Ctrl-E** can be used to switch back and forth between *Elizabeth* and the Script Editor (without changing the contents of either).

File | Treat input as script command
Treats the typed input as though it were a script command to be executed rather than user input to the system. This option (which has a keyboard shortcut of **F1**) is particularly useful when you are familiarising yourself with how the script commands work, or debugging a script which is causing problems. Note that the symbol '|' can be used to stand for a line break when you are simulating the effect of a script command which extends over several lines (which is essential if you wish to include associated actions as described in §4.2 on 'Dynamic Script Processing'.

File | Save current dialogue
Brings up a 'save file' selection dialog box, inviting you to choose a name under which the conversational dialogue that has occurred since the last restart should be saved. To avoid confusion with script files, conversational dialogues are usually saved with the filename extension `doc` (so they can straightforwardly be opened in *Word*), though they are in fact simple text files. The first line of the file will look something like this:

```
<File=My Scripts\Elizabeth.txt><RandomSeed=62850215>
```

which stores the name of the script file used to generate the conversation, and also the value of the internal random number generator 'seed' (to enable the conversation to be 'replayed' if desired).

File | Font for dialogue display/print
Brings up a font selection dialog box for choosing the font used in the 'Dialogue' tab and for dialogue printing.

File | Print current dialogue

Prints the current dialogue (i.e. the conversational dialogue that has occurred since the last restart and which is displayed in the 'Dialogue' tab). To change the font used for display and printing, select 'Font for dialogue display/print' (above).

File | Open saved dialogue
Brings up a 'load file' selection dialog box, inviting you to choose a conversational dialogue file (usually with the filename extension `doc`) which you can then 'replay'. Assuming that the script file named in the first line of the chosen dialogue is available, the system with be completely re-initialised, that script file will be read in, and a new conversation will be started with the internal random number generator 'seeded' to the same value that was used to generate the saved conversational dialogue. This means that if you supply the same inputs to the conversation (see below), then the system's responses should also be identical (i.e. whenever a random choice is made, it should be the same random choice), which can be invaluable when developing and testing a Script.

File | Take input from saved dialogue
Having opened a saved dialogue (as above), this option (which has a keyboard shortcut of **F2**) will select in turn the user inputs from that dialogue, enabling you to replay the entire dialogue (assuming that the script file has not been changed in the meantime).

File | Close saved dialogue
Closes the dialogue file (this will happen automatically once you have exhausted the inputs from it).

File | Quit
By default, quits from the system after a short delay, during which a 'quitting message' will be displayed if any have been defined. However if an 'exit message' has been defined but no quitting message, then this exit message will be displayed instead, and quitting will not be permitted. See §2.2, 'Simple Message Selection Commands', for the definition of both quitting and exit messages.

1.2.2 The View Menu

The View menu provides various settings affecting the system's 'look and feel' – its visual interface and speed of response – rather than its algorithmic processing. These settings can be set within a script by means of '/V directives', using the last item in the menu (see §6.5 for a complete list).

View | Open Current Settings Panel
Displays the Current Settings Panel, which shows the current state of the various settings whose initial values are set by the View and Processing menus, and by the RIC (recursion/iteration/cycling) Control Panel. These settings can be changed by directives within scripts (as explained below), without affecting the menu and Control Panel settings, so it is useful to be able to see, at any point, what the current state is.

View | Show All analysis tables
View | Transformation tables only
View | Hide analysis tables
The system provides comprehensive analysis tables to enable you to examine the details of the stored Script and to 'trace' the various steps of internal processing by which responses are generated. However when you are running the program for others to converse with, you will probably prefer to hide these tables. The intermediate option is to view only the tables relevant to handling input, keyword, output, and final transformations, and to hide those handling the simpler message types (i.e. welcome, quitting, exit, void input and no-keyword), since this reduces the clutter while focusing attention on those aspects of the Script that are most likely to need careful attention.

View | Instant response
View | Fast typing
View | Medium typing
View | Slow typing
When developing a Script, you will probably want the program to respond instantly to save time, but when you are running it to simulate a real conversation, it can be far more psychologically effective if it appears to be typing its response in 'real time'. Choose the speed that suits you best, balancing your patience against realism.

View | Conversational settings
View | Development settings
View | Normal settings
These three options are designed to allow you to switch between appropriate 'conversational' and 'development' settings at a single click. The former will hide the analysis tables (see above) and choose medium typing speed (or fast/slow if one of these has been previously selected while in Conversational mode); the latter will choose instant response and show all the analysis tables (or transformation tables only if this has been previously selected in Development mode). By default, the system starts up in a compromise 'Normal' mode that makes both its typing and its analysis features apparent, with fast typing and analysis tables visible.

View | Insert /View settings in script file
Inserts '/V directives' (e.g. '/V Instant Response') into the script file so that whenever it is run, the settings currently specified in the View menu will be adopted automatically (see §6.5 for a complete list).

1.2.3 The Processing Menu

The Processing menu provides settings affecting how inputs are processed into outputs by the system. These can be set within a script by means of '/P directives', using the last item in the menu (see §6.5 for a complete list).

Processing | Check Final punctuation
If selected (as it is by default), this ensures that each initial input and each final output ends with appropriate punctuation, by adding a full stop if necessary (i.e. unless there is already a full stop, an exclamation mark, or a question mark). For details of when this is done in the processing sequence, see §2.3, 'The Input/Keyword/Output/Final Transformation Process'.

Processing | Sequential responses by default
Processing | Randomised responses by default
These enable you to choose whether the responses from keyword sets and the various 'fixed' messages (such as void input and no-keyword responses) should be chosen strictly sequentially (in index order) or randomly (except for the avoidance of immediate repetition where possible). This setting can be overridden by defining the various messages using the code '!' to impose sequentiality (e.g. 'K!') or '?' to impose randomness.

Processing | Echo active text if no keyword used
Processing | Blank active text if no keyword used
If no keyword is recognised in the text, generating an appropriate response, the system can either 'echo' back the active text or delete it. The latter option is particularly useful if input text is potentially being separated into clauses that are treated independently, after which you may well wish only the recognised parts to play a role.

Processing | Upper case output only
Processing | Lower case permitted
The system usually produces only upper-case output, since this straightforwardly ensures consistency by obliterating all the case distinctions that play a major role in the operation of transformations (as explained in §5.1 on 'Capitalisation and Transformations'). However you may prefer to override this if you developing a Script and wish to see the 'raw' output, or alternatively if you have made use of the `[UPPER:]` and `[LOWER:]` functions (explained in §6.2 on 'Built-In String Functions') to produce your own preferred style of output.

Processing | Insert Processing settings in script file
Inserts '/P directives' (e.g. '/P Final punctuation ON') into the script file so that whenever it is run, the settings currently specified in the Processing menu will be adopted automatically (see §6.5 for a complete list).

1.2.4 The Control Menu

Control | Open RIC (recursion/iteration/cycling) Control Panel
Displays the Recursion/Iteration/Cycling Control Panel, which provides settings affecting recursion, and the repetition and sequencing (iteration/cycling) of input, output, and final transformations. For details see §3.2, 'Recursion and Text Splitting', and §5.2, 'Iteration and Cycling of Input/Output/Final Transformations'. For explanation of the *match algorithm limit* and *memory checking limit*, also set through this Panel, see §3.3.1 and §6.1.4 respectively. The next menu item enables all of these settings to be specified within a script.

Control | Insert RIC /Control settings in script file
Inserts '/C directives' (e.g. '/C Recursion OFF') into the script file so that whenever it is run, the settings currently specified in the RIC Control Panel (including the *match algorithm limit* and *memory checking limit*) will be adopted automatically (see §6.5 for a complete list).

1.2.5 The Analysis Menu

The Analysis menu controls the information that is made available for viewing and analyzing the internal processing of the system:

Analysis | Show hidden terms
When input/keyword/output/final transformations are read from the Script, they are modified internally in the way described at the beginning of §5.3, 'Technical Note on Pattern Matching'. Selecting this option makes the analysis tables display the full internal form of these transformations, rather than the simpler form in which they are usually specified. This will mainly be of interest to those who are using the system to investigate pattern-matching behaviour and the system's inner workings, rather than for development of a conversational Script.

Analysis | Show conditions in main tables
Script commands can be defined in such a way that they apply only if certain specified conditions are fulfilled – for explanation, see §4.5 on 'Defining and Using Conditional Commands'. Selecting this menu option makes such conditions appear within the body of the analysis tables so that they are immediately viewable (with a ':' separating the condition from the main command string); otherwise, it is necessary to click on the '[Click]' entry at the end of the relevant table to bring up a box that shows such conditions (if any conditions have been used within your Script, this column of each table will then be labelled 'Cond/Act' rather than 'Action').

Analysis | Trace input transform matching
Analysis | Trace keyword matching
Analysis | Trace output transform matching
Analysis | Trace final transform matching
Analysis | Trace condition evaluation
Analysis | Deep match tracing in these cases

The most complicated aspect of *Elizabeth's* processing is the operation of the matching algorithm, and if you find that your Script is not working as expected, there is a fair probability that the reason will lie here. Selecting these options makes the trace display include every entry and exit from the main matching routine, suitably indented to indicate their recursive structure and showing in each case the relevant entry parameters and the exit result. The operation of the matching algorithm within input, keyword, output, and final transformations, and evaluation of conditions, can be selected independently (in any combination), to make it easier to focus on whichever matching processes may be of particular interest or concern. The final option displays, or suppresses, deeper levels of the matching algorithm.

1.2.6 The Help Menu

The first item in the Help menu displays this document in PDF form (assuming that it is present in the appropriate directory). The rest of the menu provides classified selections of illustrative scripts, which will be loaded automatically into the Script Editor. Note, however, that the new Script will *not* be run within *Elizabeth* unless you transfer it by selecting the appropriate File menu option or typing **Ctrl-T**; hence you can use this method for examining an illustrative script while running your own without interference. When selected from the Help menu, scripts are copied from the 'Illustrative Scripts' subdirectory into the Script Editor, and if they are then directly saved from the editor or run, they are saved into and run from the *Elizabeth* root directory, so the original versions are unaffected by any subsequent editing.

1.3 The Script Editor

The Script Editor is essentially a simple plain text editor – accessed through *Elizabeth's* File menu (see §1.2.1) – with standard commands in its own File, Edit and View menus (such as *New*, *Open*, *Print*, *Save*, *Save As*, *Close*, *Cut*, *Copy*, *Paste*, *Find/Replace*, *Font*, and *Wordwrap*) but which is also provided with four special commands to simplify its interaction with the *Elizabeth* system:

File | Load script from Elizabeth into Editor
Loads into the Script Editor whichever script file *Elizabeth* is currently using.

File | Restart Elizabeth after saving
Saves the file which is currently being edited and then restarts *Elizabeth* with whichever script file *Elizabeth* is currently using. This current script file need not be the same as the file being edited, a possibility which is important when you are editing a component file to '#INCLUDE' within a complex script (for which see §2.1.2 on 'Script Layout and Modularisation'). This command has a keyboard shortcut of **Ctrl-R** (for 'restart' or 'run').

File | Transfer script into Elizabeth after saving
Saves the file which is currently being edited and then restarts *Elizabeth* with this same script file. **Ctrl-T** provides a keyboard shortcut (which if used from *Elizabeth*, has the reverse effect of transferring the current script into the Script Editor).

File | Switch back to Elizabeth
Switches control directly back to *Elizabeth*. The keyboard shortcut of **Ctrl-E** is provided, so this can be used to switch back and forth between *Elizabeth* and the Script Editor (without affecting the contents of either).

1.4 Illustrative Script and Conversation

Here are a simple Script and resulting conversation that bring together and illustrate most of the features explained in §2.1, 'Introduction to Script Commands', and the sections referenced by it (the line numbers are not part of the Script, but are inserted next to all of the non-comment lines to ease reference to it). Obviously the user input has been chosen to fit well with the Script, so no claim is made regarding the plausibility of the dialogue or the Script's robustness in response to other input. Note that the Script is provided as the file `Elizabeth.txt` (and also `EOriginal.txt`), to facilitate your own experimentation with it. You can view the various stages of internal processing, to see exactly what is going on, by clicking on the 'Trace' tab in *Elizabeth's* display.

Note from the numbering that some lines (namely 4, 19-23, and 35-6) are 'missing' – to find them see §2.4, 'Phrase Memorisation and Recall', and §3.2, 'Recursion and Text Splitting'. Each of these shows how this Script can easily be extended further, the first to enable user-input phrases to be memorised for recall later in the conversation, and the second to enable multiple sentences to be dealt with separately.

1.4.1 The Script

```
   /P Randomised responses
   / The Script begins with Welcome, Void, and No-Keyword responses:
1  W HELLO, I'M ELIZABETH. WHAT WOULD YOU LIKE TO TALK ABOUT?
2  V CAN'T YOU THINK OF ANYTHING TO SAY?
3  N TELL ME WHAT YOU LIKE DOING.
   /
   / Next come the Input transformations:
5  I mum => mother
6  I dad => father
   /
   / Then the Output transformations:
7  O i am => YOU ARE
8  O you are => I AM
9  O i => YOU
10 O me => YOU
11 O you [] => ME
12 O you => I
13 O my => YOUR
14 O your => MY
15 O myself => YOURSELF
16 O yourself => MYSELF
17 O I IS => I AM
18 O YOU IS => YOU ARE
   /
   / And finally four groups of Keyword transformations:
24 K I THINK [phrase]
25  R WHY DO YOU THINK [phrase]?
   /
26 K MOTHER
27 K FATHER
28  R TELL ME MORE ABOUT YOUR FAMILY.
29  R ARE YOU THE YOUNGEST IN YOUR FAMILY?
30  R WHAT DO YOU REMEMBER MOST ABOUT YOUR CHILDHOOD?
   /
31 K [phrase1] IS YOUNGER THAN [phrase2]
32  R SO [phrase2] IS OLDER THAN [phrase1].
   /
33 K I LIKE [word]ING
34  R HAVE YOU [word]ED AT ALL RECENTLY?
```

1.4.2 The Conversation

```
HELLO, I'M ELIZABETH. WHAT WOULD YOU LIKE TO TALK ABOUT?
```
(the Welcome message)

```
My mum is having a hard time.
TELL ME MORE ABOUT YOUR FAMILY
```
(line 5 changes 'mum' into 'mother', and this is matched at line 26; hence one of lines 28, 29, and 30 is randomly chosen – note that random rather that sequential choice is specified in the very first line of the script starting with '/P' – settings like this are most easily inserted using the last item in the Processing menu, which automatically inserts the currently selected processing settings)

```
There's just her, dad and my brother
WHAT DO YOU REMEMBER MOST ABOUT YOUR CHILDHOOD?
```
(line 6 changes 'dad' into 'father', and this is matched at line 27; hence either line 29 or 30 is randomly selected for display – line 28 is ruled out because it was the last response to be selected from this group)

```
My father and brother were always quarrelling.
ARE YOU THE YOUNGEST IN YOUR FAMILY?
```
('father' is matched at line 27; hence either line 28 or 29 is randomly selected for display – line 30 is ruled out because it was the last response to be selected from this group)

```
My brother is younger than me.
SO YOU ARE OLDER THAN YOUR BROTHER.
```
(line 31 matches, so line 32 then generates the response 'SO me IS OLDER THAN my brother'. Output substitutions then transform this in turn to 'SO YOU IS OLDER THAN my brother' (line 10), 'SO YOU IS OLDER THAN YOUR brother' (line 13), and finally 'SO YOU ARE OLDER THAN YOUR brother' (line 18).

```
That's right.
TELL ME WHAT YOU LIKE DOING.
```
(no keyword match is found, so line 3 is used as the system's response).

```
I like walking.
HAVE YOU WALKED AT ALL RECENTLY?
```
(line 33 matches, hence 'walk' is substituted in line 34 to generate this response).

```
No, not recently.
TELL ME WHAT YOU LIKE DOING.
```
(no keyword match is found, so line 3 is used as the system's response).
[user just presses 'Enter']

```
CAN'T YOU THINK OF ANYTHING TO SAY?
```
(the void input response, defined at line 2).

```
I think you are just repeating yourself.
WHY DO YOU THINK I AM JUST REPEATING MYSELF?
```
(line 24 matches, so line 25 then generates the response 'WHY DO YOU THINK you are just repeating yourself?'. Output substitutions then transform this in turn to 'WHY DO YOU THINK I AM just repeating yourself?' (line 8) and then 'WHY DO YOU THINK I AM just repeating MYSELF?' (line 16).

```
I think you have a stupid computer program controlling you.
WHY DO YOU THINK I HAVE A STUPID COMPUTER PROGRAM CONTROLLING ME?
```
(line 24 matches, with line 25 then generating the response 'WHY DO YOU THINK you have a stupid computer program controlling you?'. Output substitutions then transform this in turn to 'WHY DO YOU THINK you have a stupid computer program controlling ME?' (line 11), and finally 'WHY DO YOU THINK I have a stupid computer program controlling ME?' (line 12). Notice how the word 'you' is substituted by 'ME' if it occurs at the end of the active text, but by 'I' otherwise – this is because the output substitution at line 11 is dealt with before line 12, but it matches only those occurrences of 'you' that are followed by nothing.)

1.5 Weizenbaum's Original DOCTOR Script

Selecting this option from the Help menu loads into the editor a script which is almost functionally identical to Joseph Weizenbaum's original 'Doctor' script, as listed on pp. 44-5 of the article 'ELIZA – A Computer Program For the Study of Natural Language Communication Between Man And Machine', *Communications of the ACM*, January 1966, pp. 36-45. The only functional difference is to correct an error in Weizenbaum's article, since the dialogue quoted on pp. 36-7 of the article requires one set of responses to be differently ordered from the listed script. Given the fame of that dialogue (included in the *Elizabeth* package as file `WeizDoctor.doc`), it seemed better to adapt the script than disable the dialogue.

Having loaded and run the `Weizenbaum.txt` script, the Weizenbaum dialogue can be 'replayed' by using 'Open saved' dialogue from the File menu and selecting to the file `WeizDoctor.doc` in the 'Illustrative Scripts' directory – this should report that 15 dialogue inputs have been loaded. Now, in response to *Elizabeth*'s outputs, instead of typing an input, press **F1**. This should enter the text 'Men are all alike' (the first input in Weizenbaum's famous dialogue), after which clicking 'Enter' should generate the response 'IN WHAT WAY?' (as processed by *Elizabeth* using the DOCTOR rules – inspecting the trace will show what is happening). Repeatedly pressing **F1** when any input is required should then continue generating the entire dialogue:

```
   HOW DO YOU DO. PLEASE TELL ME YOUR PROBLEM.
Men are all alike.
   IN WHAT WAY?
They're always bugging us about something or other.
   CAN YOU THINK OF A SPECIFIC EXAMPLE?
Well, my boyfriend made me come here.
   YOUR BOYFRIEND MADE YOU COME HERE?
He says I'm depressed much of the time.
   I AM SORRY TO HEAR YOU ARE DEPRESSED.
It's true. I am unhappy.
   DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY?
I need some help, that much seems certain.
   WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP?
Perhaps I could learn to get along with my mother.
   TELL ME MORE ABOUT YOUR FAMILY.
My mother takes care of me.
   WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU?
My father.
   YOUR FATHER?
You are like my father in some ways.
   WHAT RESEMBLANCE DO YOU SEE?
You are not very aggressive but I think you don't want me to notice
that.
   WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE?
You don't argue with me.
   WHY DO YOU THINK I DON'T ARGUE WITH YOU?
You are afraid of me.
   DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU?
My father is afraid of everybody.
   WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR FATHER?
Bullies.
   DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR BOYFRIEND MADE
   YOU COME HERE?
```

It is instructive to read Weizenbaum's article, whilst consulting the *Elizabeth* script that generates this dialogue. Unless you are very familiar with the programming language *LISP*, the *Elizabeth* script – though not entirely simple – is a great deal easier to understand!

1.6 Self-Teach Exercises

There are five exercises, designed to be progressively more complex, and each introducing new features of the scripting language. In each case you are given a dialogue and your task is to write a script file which produces the behaviour shown in the dialogue. By the time you have completed Exercise 5, you should be ready to start producing a complete dialogue script of your own from scratch.

Exercise 1: Basic keyword-response script

For this exercise, your task is to produce a simple script in which the keywords are the acronyms 'AI' and 'NLP'. The script should produce the following behaviour:

```
    HELLO, YOU CAN ASK ME WHAT SOME ACRONYMS MEAN.
what does AI mean?
    AI MEANS ARTIFICIAL INTELLIGENCE.
what about NLP?
    NLP MEANS NATURAL LANGUAGE PROCESSING.
what about CAD?
    I DON'T KNOW ABOUT THAT ONE.
but you did know AI
    AI MEANS ARTIFICIAL INTELLIGENCE.
I don't think you know many acronyms
    I DON'T KNOW ABOUT THAT ONE.
```

This exercise shouldn't require anything beyond an understanding of the Illustrative Script and Conversation (in §1.4), which probably provides the best way in to the system, because examples are often so much easier to understand than description. However if you need more detailed discussion of *Elizabeth's* mechanisms, that section provides relevant links to other parts of this document, notably §2.3.2 on simple keyword transformations, §2.2.1 on 'welcome' messages, and §2.2.4 on 'no-keyword' responses.

Exercise 2: Simple use of memory

For this exercise the keyword 'remember' in the user input should cause the phrase following it to be stored in memory. The contents of the memory will then be stated in response to the correct keyword phrase (which you need to work out). Your script should produce the behaviour shown in the following two conversations:

```
    HI, I CAN REMEMBER THINGS.
remember my password
    OK, I'LL REMEMBER MY PASSWORD.
what's the password?
    SORRY I DON'T UNDERSTAND.
remind me what my password is
    YOU ASKED ME TO REMEMBER MY PASSWORD.
remember Joseph Weizenbaum
    OK, I'LL REMEMBER JOSEPH WEIZENBAUM.
remind me
    YOU ASKED ME TO REMEMBER JOSEPH WEIZENBAUM.
remind
    SORRY I DON'T UNDERSTAND.
tell me
    SORRY I DON'T UNDERSTAND.


    HI, I CAN REMEMBER THINGS.
remind me
    SORRY I DON'T UNDERSTAND.
remember my name
    OK, I'LL REMEMBER MY NAME.
my name is Joe
    SORRY I DON'T UNDERSTAND.
```

```
remind me what my name is
   YOU ASKED ME TO REMEMBER MY NAME.
remember the armadillo
   OK, I'LL REMEMBER THE ARMADILLO.
remember my pet armadillo
   OK, I'LL REMEMBER MY PET ARMADILLO.
remind me
   YOU ASKED ME TO REMEMBER MY PET ARMADILLO.
```

Again refer to the Illustrative Script and Conversation in §1.4 and (if necessary) the other sections mentioned in connection with Exercise 1, but also §2.4 on 'Phrase Memorisation and Recall'.

Exercise 3: Transformations and more memory

This exercise requires you to modify the script you did for Exercise 2 (but save your existing script first under a separate name, so you don't lose it). Your Exercise 2 script should produce the behaviour in the last conversation shown above. For Exercise 3, modify the script so it produces the behaviour below. Note the different response in the third line, and how 'my' at various points becomes 'YOUR'.

```
   HI, I CAN REMEMBER THINGS.
remind me
   YOU ASKED ME TO REMEMBER NOTHING SO FAR.
remember my name
   OK, I'LL REMEMBER YOUR NAME.
my name is Joe
   SORRY I DON'T UNDERSTAND.
remind me what my name is
   YOU ASKED ME TO REMEMBER YOUR NAME.
remember the armadillo
   OK, I'LL REMEMBER THE ARMADILLO.
remember my pet armadillo
   OK, I'LL REMEMBER YOUR PET ARMADILLO.
remind me
   YOU ASKED ME TO REMEMBER YOUR PET ARMADILLO.
```

The help references here are the same as for Exercise 2, plus §2.3.4 on output transformations. Note that the output in the third line is relatively easy to achieve because it begins with the words 'YOU ASKED ME TO REMEMBER' (so you can achieve this by setting an initial memory within your script). But you might find it interesting to consider how you would instead produce a different form of words, such as 'YOU HAVEN'T ASKED ME TO REMEMBER ANYTHING YET'. One simple method is to use two keyword sets which have the same keyword, but where the response in the first set contains a memory, and hence only becomes available when that memory has become defined (this is illustrated in §2.4.1, 'Remembering More Than One Thing'). But you can also use the more advanced techniques from §4.2, 'Dynamic Script Processing' and §4.5, 'Defining and Using Conditional Commands').

Exercise 4: Remembering more than one thing

For Exercise 4 the script has to deal with remembering more than one thing at a time. The required behaviour is shown in the following dialogue.

```
   HI, I CAN REMEMBER A NAME AND A NUMBER.
remember the name Elizabeth
   OK, I'LL REMEMBER THE NAME ELIZABETH.
recall the number
   SORRY, I DON'T UNDERSTAND.
remember the number 42
   OK, I'LL REMEMBER THE NUMBER 42.
recall the name
   THE NAME IS ELIZABETH.
recall the number
```

```
   THE NUMBER IS 42.
what is the name?
   SORRY, I DON'T UNDERSTAND.
```

Again this exercise requires the techniques of ‘Phrase Memorisation and Recall’ (§2.4), but now augmented by the use of index codes as described in §2.4.1, ‘Remembering More Than One Thing’.

Exercise 5: Dynamically altering the script

This exercise is more challenging than the previous ones. The following conversation concerns the numbers *one* and *two*, their Roman forms (‘I’ and ‘II’) and the familiar Arabic equivalents (‘1’ and ‘2’). Note that the response to typing ‘one’ or ‘two’ depends on what has gone on before in the conversation. Also note that before Roman or Arabic has been specified, the response to ‘one’ is neither ‘I’ nor ‘1’.

```
   I KNOW BOTH ROMAN AND ARABIC VERSIONS OF ONE AND TWO.
one
   IS THAT ROMAN OR ARABIC?
roman
   OK, I'LL USE ROMAN NUMERALS.
one
   I.
two
   II.
arabic
   OK, I'LL USE ARABIC NUMERALS.
one
   1.
two
   2.
three
   SORRY NO IDEA.
roman
   OK I'LL USE ROMAN NUMERALS.
one
   I.
two
   II.
```

There are various ways to achieve the behaviour required here, and you might find it interesting to experiment with different techniques. Some just involve the saving and/or deletion of memories (as explained in §2.4, ‘Phrase Memorisation and Recall’, and §4.4, ‘Giving Direction to a Conversation’) and possibly the use of conditional commands (see §4.5, ‘Defining and Using Conditional Commands’). Others involve Dynamic Script Processing (see §4.2), such as the creation and deletion of keywords etc. while the conversation is running.

2. Basic Script Commands

2.1 Introduction to Script Commands

Unless it has been 'modularised' as described below, the Script is a single plain text file each of whose lines begins with either a letter ('W', 'X', 'Q', 'V', 'N', 'H', 'I', 'K', 'R', 'O', 'F', 'M', or 'S') or another defined symbol ('&', '<', or '/'). In every case except '<' (which indicates a command condition as explained in §4.5, 'Defining and Using Conditional Commands', and not further discussed below), this first character is followed by a space and then a specification of:

W a 'welcome' message.

Q a 'quitting' message.

X an 'exit' message.

V a 'void input' response.

N a 'no-keyword' response.

For details of all the above message types, see §2.2, 'Simple Message Selection Commands'.

H a halting message.

This message comes into play only if the system halts on exceeding the *match algorithm limit* or the *memory checking limit*, as explained in §3.3.1 and §6.1.4 respectively.

I an input transformation.

K a keyword pattern.

R a keyword response pattern – should immediately follow the corresponding keyword pattern(s).

O an output transformation.

F a final transformation.

For details of these transformations, see §2.3, 'The Input/Keyword/Output/Final Transformation Process'.

M a memorised phrase.

For details, see §2.4, 'Phrase Memorisation and Recall'.

S save text to file.

For details, see §2.5, 'Saving Text Files Dynamically'.

& an action to be performed when a message is displayed or a transformation takes place – should immediately follow a message or transformation specification or a replacement pattern.

For details, see §4.2, 'Dynamic Script Processing'.

/ a comment (in which case the entire line from the Script is ignored).

*Note that you can experiment with these commands by entering them directly from the keyboard (pressing **F1** instead of 'Enter') as well as from a Script. For details, see the relevant File menu command in §1.2.1.*

2.1.2 Script Layout and Modularisation

All spaces at the beginning of Script lines are ignored, so judicious indenting can be used to emphasise the Script structure, as is done in this document. In particular, it is good practice always to indent any '&' action-line, to show visually which message or transformation command it corresponds to.

As you progressively add commands to generate more interesting and flexible responses, you are likely to find that a single Script file soon becomes rather unwieldy. Moreover it is good practice to 'modularise' your working so that you focus on one problem at a time – for example, you might start by specifying some simple messages, then move on to perfect the design of output transformations to transpose first and second-person pronouns ('I' to 'you' and vice-versa etc.) before finally turning to consider your keyword transformations in

detail. For these reasons, the system has been designed to enable you to divide up your Script into as many files as you wish. In the situation just described, you would simply include the directives:

```
#INCLUDE SimpleMessages.txt
#INCLUDE OutputTransforms.txt
```

within your main Script file, having saved all your simple message specifications into the file `SimpleMessages.txt`, and the perfected output transformations into the file `OutputTransforms.txt`. An '#INCLUDE' directive has exactly the same effect as copying the included file into the corresponding position within the main Script file. (And note that '#INCLUDE' directives can be used not only within that main file, but also within included files themselves, to a 'depth' of up to 20 levels.)

2.1.3 The 'Turbo Eliza' Example Files

To provide additional material for illustration and experimentation, the system includes a Script derived from the one accompanying a public domain ELIZA program, of unknown authorship, written in Turbo Pascal and dating from 1987. That program, here referred to as 'Turbo Eliza', has for many years been mounted on various FTP and Web sites across the world (as a search for the keywords 'Eliza' and 'bbschat' will reveal), and was perhaps the best public domain version available – for example the Simon Laven website at http://www.simonlaven.com (a major 'chatbot' hub since 1996) describes it as 'A brilliant Borland version for you to use at your leisure. The best Eliza on here.' The original Script file, `bbschat.net`, has been translated for the *Elizabeth* program to provide more or less identical behaviour, and is provided here in two equivalent versions. The first version contains the entire script in a single file, named `TurboAll.txt`, and is well worth examining to get an overview of how everything fits together. The second version is better suited for experimentation, being divided into modular files (such as `TurboFixed.txt`, `TurboInput.txt` etc.) that are all invoked by '#INCLUDE' directives in the main file `TurboEliza.txt`. Some of these modular files may well prove useful when developing your own Script (e.g. the '#INCLUDE My Scripts\TurboOutput.txt' directive will straightforwardly provide you with simple pronoun transposition facilities). For further details of what each file contains, see §6.4, 'Directory Usage, System Files, and Predefined Scripts'.

2.2 Simple Message Selection Commands

The commands 'W', 'Q', 'X', 'V', and 'N' all operate in a very simple manner, being used to specify 'welcome', 'quitting', 'exit', 'void input', and 'no-keyword' messages respectively. Welcome messages are used to start the conversation, quitting messages to say goodbye, exit messages to respond when the user tries to exit but is not to be permitted to do so, void input messages to provide a response when the user simply presses 'Enter' without first typing any input (i.e. fails to contribute to the conversation on his or her turn), and no-keyword messages when the user's input contains no identified keywords. Unlike keyword responses, which can employ fairly sophisticated pattern matching to reflect the user's input, these four kinds of message are all relatively 'fixed' (though they can involve substitution of memorised phrases, as explained in §2.4, 'Phrase Memorisation and Recall'. Three other general points apply to them:

If no appropriate message is defined

If no void-input response has been defined, then the system will use the built-in message 'I CAN'T THINK OF ANYTHING TO SAY' in response to a void input. Likewise if no 'welcome' message has been defined, then this same built-in message will be used to start any dialogue. So it's worth remembering that 'I CAN'T THINK OF ANYTHING TO SAY' usually indicates an inadequate Script. The behaviour in the case of 'no-keyword' messages is different – here, if there are no such messages defined, the active text (i.e. in most cases, the user's input) is either 'echoed' (i.e. repeated back) or deleted entirely ('blanked'), depending on the relevant setting in the Processing menu. Finally, the presence of 'quitting' and 'exit' messages itself indicates whether the user is to be allowed to quit or exit from the system in certain circumstances (see below), and if no such message has been defined, there is no need for a built-in default.

If more than one appropriate message is defined

If more than one of any of these five types of message is specified, then whenever such a message is needed, a choice will be made from those available, but the method of choice will depend on the relevant setting in the Processing menu. If 'Sequential responses by default' is selected, then the available messages

will be chosen cyclically in order of definition (or strictly, order of index code), starting with the first. If, on the other hand, 'Randomised responses by default' is selected, then they will be chosen randomly *except* that the system will attempt to make a different random choice from the last one previously made. So if, for example, you define two or more void input or no-keyword responses, you can be sure either way that the same such response will never be made twice in a row. (Note also that this behaviour can be fine-tuned for particular types of message within the script – for example, '`N? ANOTHER RESPONSE`' will select randomised choice for no-keyword responses, while '`N! ANOTHER RESPONSE`' will select sequential choice for them.)

If messages are defined as self-deleting

If any of these messages is defined by means of a command that has '\' as the first character, for example:

```
\N CHANGING THE SUBJECT, DO YOU LIKE SPORT?
```

then when the message has been used once, it will automatically be deleted. This provides a very simple mechanism to avoid undesirable repetition, but be careful not to define *all* your no-keyword messages (etc.) as self-deleting, since otherwise *Elizabeth* may be reduced to using the default: 'I CAN'T THINK OF ANYTHING TO SAY'. (See §4.2.4, 'Self-Deletion', for more details of this facility.)

2.2.1 Welcome Messages

A line in the Script starting with 'W' specifies a welcome message, for example:

```
W HELLO, I'M ELIZABETH. WHAT WOULD YOU LIKE TO TALK ABOUT?
```

When the system starts up, one of these defined welcome messages will be chosen at random and displayed. So if you specify only one such message, then every conversation will start in the same way, but if you want more variety, simply add more such messages.

2.2.2 Quitting Messages

A line in the Script starting with 'Q' specifies a quitting message, for example:

```
Q GOODBYE!  DO COME BACK SOON.
```

When you quit from the system (using the 'Quit' option from the File menu), one of these defined quitting messages will be chosen at random and displayed for a short time before the system shuts down. So if you specify only one such message, then every conversation will end in the same way, but if you want more variety, simply add more such messages. See below for how quitting and exit messages interact.

2.2.3 Exit Messages

A line in the Script starting with 'X' specifies an exit message, for example:

```
X PLEASE FINISH THE DIALOGUE BEFORE QUITTING.
```

The presence of such a message modifies the system's behaviour in respect of 'exiting' (i.e. closing the system window by clicking on the '×' at the top right corner) and 'quitting' (i.e. selecting 'Quit' from the File menu). Quitting messages and exit messages interact as follows, with the most distinctive option here being the last of the four.

If no quitting message and no exit message has been defined,

then the user is allowed both to quit and to exit. In either case the program will terminate immediately.

If a quitting message is defined, but no exit message,

then the user is allowed both to quit and to exit. In either case a quitting message (e.g. 'BYE!') will be used as a response before the program terminates.

If both quitting and exit messages are defined,

then the user is allowed to quit, but not to exit. A quitting message (e.g. 'BYE!') will be used as a response before quitting, while if the user attempts to exit, this will be prevented and an exit message (e.g. 'Please use "Quit" from the File menu') displayed in an information box.

If an exit message is defined, but no quitting message,

then both quitting and exiting will be prevented, with an exit message (e.g. 'PLEASE FINISH THE DIALOGUE') being displayed in an information box in both cases.

2.2.4 Void Input Responses

A line in the Script starting with 'V' specifies a void input response, for example:

```
V CAN'T YOU THINK OF ANYTHING TO SAY?
```

Whenever the user enters a 'void input' (i.e. by pressing or clicking on 'Enter' without typing any text), then unless this void input is dealt with by an input or keyword transformation (see below), one of these defined void input responses will be chosen at random and displayed. It is a good idea to specify a number of such responses, to avoid obvious repetition if the user enters several void inputs during the course of the conversation.

2.2.5 No-Keyword Responses

If no keyword is found within the active text, then one of the defined *no-keyword responses* is randomly chosen to replace it. These are specified by lines in the script starting with 'N', for example:

```
N I'M NOT SURE THAT I FULLY UNDERSTAND YOU.
```

2.2.6 Adaptive Void Input and No-Keyword Behaviour

Void input and no-keyword reponses do not pass through the stage of processing by output transformations, since they are usually assumed to be defined appropriately for immediate output without further substitution. (Though they are processed by final transformations, which can be useful for advanced users.) Note, however, that both types of response can be simulated using keyword transformations, for example by including as your last two keyword sets:

```
K []
R THIS WILL ACT LIKE A VOID INPUT RESPONSE
K [X]
R THIS WILL ACT LIKE A NO-KEYWORD RESPONSE
```

These will 'sweep up' any text left over by previous keyword sets, ensuring that some keyword will *always* match, and hence make redundant any void input or no-keyword responses. For a use of this technique, see §3.2, 'Recursion and Text Splitting'.

2.3 The Input/Keyword/Output/Final Transformation Process

Leaving aside welcome, quitting and exit messages, and also recursion, the typical method by which the user's input is transformed into an appropriate conversational output is as follows:

1. The *active text* (i.e. the text currently being processed) is set equal to the user's input, with all letters converted into lower case and spaces inserted between words and punctuation. If the input is not void (i.e. empty), and does not already end with either a full stop, an exclamation mark or a question mark, then a full stop is added at the end (although this behaviour can be suppressed by deselecting 'Check final punctuation' in the Processing menu).
2. Any defined *input transformations* are applied, in turn, to the active text.
3. The active text is searched, in turn, for each defined *keyword pattern*, until one of these searches succeeds or all have failed.

4 If a keyword pattern is found within the active text, then the active text is replaced by a chosen member of the *response set* corresponding to that keyword pattern (this choice can be sequential or randomised according to the relevant setting in the Processing menu, exactly as with the Simple Message Selection Commands in §2.2, with the additional feature that '`K! AKEYWORD`' will force just the relevant keyword set to be sequential, and '`K? AKEYWORD`' randomised).

5. Any defined *output transformations* are applied, in turn, to the active text.

6a If no keyword pattern has been found within the active text, then unless the user's input text was void (i.e. empty), the active text is replaced by a chosen member of the set of *no-keyword* responses.

6b If no keyword pattern has been found, and the user's input text was void (i.e. empty), then the active text is replaced by a chosen member of the set of *void input* responses.

7. Any defined *final transformations* are applied, in turn, to the active text.

8. The resulting form of the active text is converted to upper case and is then output as the program's conversational response, with punctuation automatically being spaced correctly, and a full stop added at the end if the active text is not void and no suitable punctuation is present already (as at the first stage, this full stop checking can be suppressed, by deselecting 'Check final punctuation' in the Processing menu).

9. After the processing of the active text is complete, any pending dynamic actions are performed in sequence, as explained in §5.4, 'Sequencing and Timing of Dynamic Commands'.

10. Finally, any messages, transformations, keywords or responses that have been defined as *self-deleting*, and have been used in the processing described above, are deleted. Note that any keyword or response, and any input, output or final transformation, can easily be made self-deleting by preceding the relevant command with '\', in a similar way to the no-keyword example given in §2.2, 'Simple Message Selection Commands' (and see §4.2.4, 'Self-Deletion' for details)

2.3.1 Input Transformations

A line in the Script starting with 'I' specifies an input transformation – a transformation that takes place at stage 2 as outlined above (and hence prior to keyword, output, and final transformations). Thus for example the following:

```
I dad => father
```

means that whenever the user inputs a sentence containing one or more occurrences of the word 'dad', these will be transformed at this early stage of processing into 'father'. Note that in general it is best to use lower-case letters for both the left-hand and the right-hand sides of these transformation rules (the reason for this is explained in §5.1, 'Capitalisation and Transformations').

In simple cases, the input transformations are applied in turn, in the same order that they are specified in the Script. For further details, and an explanation of how the sequencing can be controlled, see §5.2, 'Iteration and Cycling of Input/Output/Final Transformations'.

2.3.2 Keyword Transformations I – Simple Replacement

Keyword transformations take place after any input transformations, but before any output or final transformations. Unlike these other types of transformation, which are 'deterministic' (i.e. the same words in the active text will always be transformed into the same result), keyword transformations usually involve random selection in much the same way as void input and no-keyword responses (described in §2.2, 'Simple Message Selection Commands'). Any keyword transformation involves two main elements:

(a) a set of *keyword patterns* that the system attempts to 'match' against the active text;

(b) a set of *keyword responses* linked to each set of keyword patterns.

If one of the keyword patterns is indeed matched by the active text, then one of the corresponding set of responses is randomly selected to replace the active text.

Keyword patterns are specified by lines in the Script starting with 'K', and keyword responses by lines starting with 'R'. *Keyword patterns that are grouped together within the Script will be treated as members of the same keyword pattern set, and the corresponding keyword responses should likewise be grouped together and should*

immediately follow the keyword patterns. Suppose, for example, that the Script contains the following sequence of four lines:

```
K MOTHER
K FATHER
 R TELL ME MORE ABOUT YOUR FAMILY.
 R WHAT DO YOU REMEMBER ABOUT YOUR CHILDHOOD?
```

Then if the active text contains either the word 'mother' or the word 'father' (and no keyword from an earlier set has already been found within it), then the active text will be replaced with one of the two response sentences shown. Note that in general it is best to use upper-case letters for both the keyword patterns and keyword responses (again see §5.1, 'Capitalisation and Transformations', for explanation).

2.3.3 Keyword Transformations II – Replacement and Substitution

The examples in the previous section involve very simple keyword pattern-matching and replacement: the active text is replaced in its entirety if it contains, at any point, either of the words 'mother' or 'father'. But far more sophisticated processing can be done, best introduced by example. Suppose, then, that the Script contains the following sequence of five lines:

```
K CAN I [phrase]
K MAY I [phrase]
 R DO YOU WANT TO [phrase]?
 R WOULD IT MAKE YOU HAPPY TO [phrase]?
 R WOULD YOU [phrase] IF YOU KNEW YOU COULD?
```

The first of these lines means that the keyword transformation in question is to be triggered if the active text 'matches' the pattern 'CAN I `[phrase]`' – in other words, if it contains at some point the words 'CAN I' *followed by some phrase or other* (so the 'I' cannot be the last word of the active text, because any phrase must include at least one word). When this happens, the active text is replaced by one of the three specified keyword reponses, *with `[phrase]` being replaced by the phrase in question*. So for example the input:

```
CAN I GET A DEGREE?
```

would be matched against the keyword pattern:

```
CAN I [phrase]
```

with 'GET A DEGREE' substituted for `[phrase]`, and this might then result in any of the following three outputs depending on which of the three keyword responses is randomly chosen:

```
DO YOU WANT TO GET A DEGREE?
WOULD IT MAKE YOU HAPPY TO GET A DEGREE?
WOULD YOU GET A DEGREE IF YOU KNEW YOU COULD?
```

For further pattern matching and substitution options, see §3.1, 'Pattern Matching'.

2.3.4 Output Transformations

A line in the Script starting with 'O' specifies an output transformation – a transformation that takes place at stage 5 as outlined above (and hence after both input and keyword transformations). Thus for example the following:

```
O you are => I AM
```

means that whenever the active text at this stage contains the phrase 'you are', this will be replaced by 'I AM' – this sort of thing is typically done to ensure that second-person references by the user are turned into first-person references by the computer, and vice-versa. Note that, unlike input transformations, it is generally best to use lower-case letters for the left-hand side of these transformation rules, and upper-case for the right-hand side (again, the reason for this is explained in §5.1, 'Capitalisation and Transformations').

In simple cases, the output transformations are applied in turn, in the same order that they are specified in the Script. For further details, and an explanation of how the sequencing can be controlled, see §5.2, 'Iteration and Cycling of Input/Output/Final Transformations'.

2.3.5 Final Transformations

A line in the Script starting with 'F' specifies a final transformation – a transformation that takes place at stage 6 as outlined above (and hence after all input, keyword, and output transformations). *Final transformations are essentially identical to output transformations, except where recursion and text splitting (§3.2) are involved.* Briefly, the difference is that when the active text is split into two or more components for separate processing, the processed components are recombined *after* any output transformations have been applied to each of the components, but *before* applying any final transformations. Hence final transformations provide an opportunity to define transformations that are intended to apply to the entire recombined text just before it is output (an example where this is used to remove duplication generated from separate text components is in §3.2, 'Recursion and Text Splitting').

2.4 Phrase Memorisation and Recall

A particularly striking way of continuing a conversation, especially useful when no keyword has been identified, is to recall a relevant phrase that has previously been input by the user. The most common way of doing this is to choose phrases following the word 'my' – in the case of the Illustrative Script discussed in §1.4, such memorisation and recall can be done by including the following lines in the Script:

```
21  K [] MY [phrase]
22    & {M [phrase]}
23   R YOUR [phrase]?
```

If these had been inserted into the illustrative Script, then the conversation would instead have started:

```
HELLO, I'M ELIZABETH. WHAT WOULD YOU LIKE TO TALK ABOUT?
My mum is having a hard time.

YOUR MOTHER IS HAVING A HARD TIME?
```

This response is straightforwardly produced by lines 5, 21, and 23:

(a) 'mum' is changed to 'mother' by the input transformation at line 5;

(b) 'my mother is having a hard time' matches against the keyword pattern at line 21 (the initial '`[]`' in the keyword pattern ensures that a match takes place only if 'my' is at the beginning of the active text – see §3.1, 'Pattern Matching' for this and other pattern elements);

(c) When the match at line 21 takes place, `[phrase]` is matched to 'mother is having a hard time'; this is then inserted into the response pattern at line 23 to generate 'YOUR mother is having a hard time?'.

Meanwhile, however, line 22 memorises 'mother is having a hard time' for future use, and this can later be recalled into the conversation if the Script contains another insertion at line 4:

```
4  N DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR [M]?
```

This introduces a second possible no-keyword response (but one which will be unavailable for selection until some phrase has previously been saved). In the situation discussed here, line 4 would generate the response:

```
DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR MOTHER IS HAVING A
HARD TIME?
```

Thus the term `[M]` in line 4 stands for whatever phrase has been saved using commands of the kind in line 22, but if several phrases have been memorised during the course of a conversation, then `[M]` stands for the most recent of them. Note that you can also use `[M-1]` for the last-but-one of any saved phrases, `[M-2]` for the last-but-two, and so on; also `[M+1]` for the first of any saved phrases, `[M+2]` for the second saved, and so on.

(Strictly, the ordering is done by index codes rather than age, as explained in §4.3, 'Control of Scripts using Command Index Codes', but in practice this comes to the same thing if you don't specify such codes yourself, because then the phrases are automatically given sequential codes '001', '002', '003' etc. If you wish to save some phrases using index codes while also retaining the ability to recall others according to their age, then the easiest solution is to use index codes starting with '$' or '%', which are alphabetically earlier than the automatic numeric codes. The allowable characters, and their alphabetical sequence, are specified in §6.3, 'Alphabets and Special Symbols'.)

To make [M] responses meaningful even before the user has entered a 'my' phrase, you can include a command to memorise a suitably vague phrase within your initial Script, for example:

```
M LIFE IS DIFFICULT JUST NOW
```

which if set up with line 4 as described above could generate the response:

```
DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR LIFE IS DIFFICULT
JUST NOW?
```

2.4.1 Remembering More Than One Thing

If you want *Elizabeth* to remember a number of different things (without having to worry about which is the latest, the last-but-one etc.), then use index codes when you save and recall them, for example:

```
K MY NAME IS [phrase]
 R OK - YOUR NAME IS [phrase]
   & {Mnamecode [phrase]}

K MY ADDRESS IS [phrase]
 R OK - YOUR ADDRESS IS [phrase]
   & {Maddrcode [phrase]}

K REPORT
 R YOUR NAME IS [Mnamecode] AND YOUR ADDRESS IS [Maddrcode]

K REPORT
 R YOU HAVEN'T GIVEN ME YOUR NAME AND ADDRESS YET
```

If you look at the Memory display as you run this script and input 'My name is Joe' etc., you'll see that the name and address memories are stored under the index codes you've specified in the script (i.e. 'namecode' and 'addrcode' respectively), enabling them to be recalled using those codes when you type 'Report'. Notice also here the trick of having two keyword sets with the same keyword 'REPORT' – the second set will never be used unless the first fails to operate, which will be the case if either [Mnamecode] or [Maddrcode] is undefined. Hence if either your name or address has not yet been given, the response to 'Report' will appropriately be 'YOU HAVEN'T GIVEN ME YOUR NAME AND ADDRESS YET.'

What have been described here are only the simplest applications of phrase memorisation and recall. For more sophisticated uses, see §4.4, 'Giving Direction to a Conversation' (which gives more examples of using index codes to handle different memories) and §4.2, 'Dynamic Script Processing'.

2.4.2 Implementing Memory Arrays

In the 'Basic script commands and memory' submenu of the Help menu, there is a script named 'Implementing memory arrays' whose behaviour can be illustrated as follows:

```
Find Tom.
   TOM NOT FOUND.
Save Tom.
   TOM SAVED AS ARRAY_1.
Find Tom.
   TOM FOUND AT ARRAY_1.
Save Dick.
   DICK SAVED AS ARRAY_2
Find Harry.
   HARRY NOT FOUND.
Find Dick.
   DICK FOUND AT ARRAY_2
```

This is achieved as follows:

(a) The memory 'index' – initialised to 'Array_1' – keeps track of the next location available for definition. Each time another item is saved to the 'array' (e.g. by making memory Array_1 equal to 'tom'), 'index' is incremented using the 'inc' function (see §6.2, 'Built-In String Functions').

(b) Saving and incrementing are achieved using dynamic commands in the standard way.

(c) Typing 'Find Dick' triggers a keyword which substitutes the single string 'RETRIEVE', while dynamically setting the memory 'searchindex' to 'Array_1' and 'searchstring' to 'Dick'.

(d) 'RETRIEVE' can trigger any one of three keywords, the first two of which are conditional:

(d1) If 'searchindex' is identical to 'index', then the response is 'NOT FOUND';

(d2) If the memory whose code is 'searchindex' is identical to 'searchstring', then the item has been 'FOUND' at the position given by 'searchindex';

(d3) Otherwise, 'searchindex' is incremented, and the single string 'RETRIEVE' is recursed (by '{RETRIEVE}', see §3.2 for more detail), effectively searching the next item in the array.

This gives a simple illustration of how memories can be organised and searched, and the same techniques could easily be extended to multiple arrays and other operations. One nice exercise is to work out how to perform deletion (perhaps either by name or by index), and how then to allow insertion within vacant 'spaces' whilst still maintaining full search of the saved items.

2.4.3 Automatic Memory of the Preceding Dialogue

In addition to the user-defined facility for memorisation described above, *Elizabeth* automatically remembers all previous inputs and outputs in the current dialogue. The syntactic options here are very similar to those for other memorised phrases, except that the relevant letters are 'I' and 'O' rather than 'M', for example:

`[I]`	The latest input to *Elizabeth* (i.e. the 'current' user input)
`[I-2]`	The last-but-two input to *Elizabeth* (if there have been three or more)
`[I+3]`	The third input to *Elizabeth* (if there have been three or more)
`[O]`	*Elizabeth's* latest output
`[O+1]`	The very first output from *Elizabeth* (i.e. the initial welcome message)

2.5 Saving Text Files Dynamically

The `S` script command can be used to save text during a dialogue, for example to auto-record a dialogue while it's underway, or to record user input from a questionnaire. The following example illustrates this (in a context similar to one in §2.4, 'Phrase Memorisation and Recall', where the name is saved to memory):

```
K MY NAME IS [phrase]
 R OK - YOUR NAME IS [phrase]
   & {Smyfile.txt NAME: [phrase]}
```

If the user inputs 'my name is fred bloggs', the response 'OK - YOUR NAME IS FRED BLOGGS.' is triggered, and this is accompanied by the action:

```
Smyfile.txt NAME: FRED BLOGGS
```

which simply instructs *Elizabeth* to append the text 'NAME: FRED BLOGGS' to the file `myfile.txt` (creating the file first if it does not already exist). The `S` script command is fairly flexible, as illustrated by the following possibilities:

```
Smyfile.txt MYTEXT
```

Appends 'MYTEXT' to the file `myfile.txt`, where the filename is interpreted as being specified relative to the main *Elizabeth* directory.

```
Smyfile.txt MYTEXT\n
```

This performs the same appending operation to `myfile.txt`, except that '\n' inserts an end-of-line in the file immediately following 'MYTEXT'.

```
Smyfile.txt\ MYTEXT
```

Here the '\' immediately following the filename means that the file will be newly created with the specified text as its only content, rather than having the text appended to its current contents.

```
Smyfile.txt\
```

Where the filename is immediately followed by '\', but without any text being specified, this means that the named file (if it exists) will be deleted.

2.5.1 Using Conditions and Actions in Saving Commands

Conditions and further actions can be specified in the `S` command, using the same sort of syntax that applies to other script commands (as discussed in §4.5, 'Defining and Using Conditional Commands', and §4.2, 'Dynamic Script Processing'). Again their behaviour is most easily explained by example:

```
K MY NAME IS [phrase]
 R OK - YOUR NAME IS [phrase]
   & {<[Mnameneeded]>: Smyfile.txt NAME: [phrase]}
```

This is identical to the first example above, except that the condition `<[nameneeded]>:` means that the appending of text to `myfile.txt` will be performed only if the memory `[Mnameneeded]` has been defined.

```
K MY NAME IS [phrase]
 R OK - YOUR NAME IS [phrase]
   & {Smyfile.txt NAME: [phrase]
      & {Mnamesaved [phrase]}}
```

Here an action has been specified, to be performed when the text appending operation takes place. Note here that the action (in this case the saving of a memory) will only be performed if the file operation is *successful*; hence if for any reason it fails, for example through some filing system error, then the action specification will be ignored. This feature is particularly useful when conditions and actions are combined in the following sort of way:

```
K MY NAME IS [phrase]
 R OK - YOUR NAME IS [phrase]
   & {<[Mnameneeded]>: Smyfile.txt NAME: [phrase]
                       & {Mnameneeded\}}
```

Here an attempt is made to save the text only if the memory `[Mnameneeded]` is defined, but then if the saving attempt is successful, that memory is immediately deleted. Hence this mechanism can be used to ensure that the test in question is saved to file once only.

3. Pattern Matching and Recursion

3.1 Pattern Matching

In all of its transformation processes, the system is able to handle a wide range of text-matching patterns, including any of the following:

3.1.1 Individual (Non-Space) Characters

`[letter]`	Any single 'letter' ('A' .. 'Z', 'a' .. 'z', plus the hyphen and apostrophe).
`[digit]`	Any single digit ('0' .. '9').
`[alphanum]`	Any single alphanumeric character (i.e. either a 'letter' or a digit).
`[char]`	Any single character which is not a punctuation mark.
`[,]`	Any comma, semicolon, or colon.
`[.]`	Any full stop, exclamation mark, or question mark.
`[;]`	Any single punctuation mark.

3.1.2 Contiguous Sequences of (Non-Space) Characters (i.e. 'Items')

`[word]`	Any contiguous sequence of 'letters' ('A' .. 'Z', 'a' .. 'z', plus the hyphen and apostrophe).
`[number]`	Any contiguous sequence of digits ('0' .. '9').
`[term]`	Any contiguous sequence of alphanumeric characters (i.e. 'letters' and digits).
`[string]`	Any contiguous sequence of any characters other than punctuation marks.

3.1.3 Sequences of Complete 'Items' (Including Separating Spaces)

`[phrase]`	Any sequence of one or more complete items consisting entirely of 'letters' (which therefore cannot include any digits, punctuation, or other non-letter symbols).
`[expression]`	Any sequence of one or more complete items consisting entirely of alphanumeric characters (i.e. 'letters' and digits).
`[formula]`	Any sequence of one or more complete items which do not include any punctuation marks (but can include 'letters', digits, and non-punctuation symbols such as '$' and brackets).
`[x]`	The most flexible pattern available (hence the use of the general variable 'x') – any sequence of one or more complete items (of any kind) and/or punctuation.
`[bracket]`	Like `[x]`, except that this will not match any sequence that contains an 'unpaired' curved or angle bracket (i.e. a curved or angle bracket whose corresponding bracket is not also contained in the matched text). This term type is provided specifically for dealing with grammars and other formal syntactic processing, for which see the sections on Implementing Grammatical Rules (§4.1) and Implementing Propositional Logic (§4.6).
`[!]`	Any sequence of punctuation marks.

3.1.4 Text Beginning and End

`[]` Nothing – reserved for use at the beginning or end of a pattern to signify that it should match the active text only if the pattern extends to the very beginning or end of that text. *(The end of the text here can include the final punctuation if any is present, but need not – for details, see §5.3, 'Technical Note on Pattern Matching')*. By itself, this pattern matches only the null text (i.e. where the active text is completely empty).

All but the last of these can be abbreviated and/or numbered, so for example `[w1]` and `[w2]` might be used for two words, `[phr1]` and `[phr2]` for two phrases (that don't include punctuation), `[x]` or `[brak]` for a sequence of words and punctuation, `[l]` for a letter, and `[;1]` for a punctuation mark (in fact, in *all* cases where something appears between the square brackets, it is only the first character which determines what kind of pattern is specified, so at the risk of causing confusion, you could for example use `[tom]` for a term). For basic illustrations of processing using `[word]` and `[phrase]`, which are perhaps the simplest of these patterns, see the Illustrative Script and Conversation in §1.4.

3.1.5 Matching Nothing

In some circumstances it can be useful to specify patterns that are able to 'match' either something or nothing at all. Suppose, for example, that when the user enters a sentence containing the word 'uni', the system is intended to respond with the same sentence followed by a question-mark, but with 'uni' replaced by 'university' for example:

```
Being at the uni is fun
BEING AT THE UNIVERSITY IS FUN?
```

It might seem that the simplest way of achieving this is with the Script commands:

```
K [phrase1] UNI [phrase2]
 R [phrase1] UNIVERSITY [phrase2]?
```

However this would fail with the input:

```
Uni is fun
```

because this does not match the specified keyword pattern, there being nothing before the word 'UNI' to match with the `[phrase1]` term. The solution is to add two question-marks within the keyword specification command:

```
K [phrase1?] UNI [phrase2?]
```

Here a question-mark means that the relevant term can be matched by something (i.e. in these cases, a phrase) but also by nothing. So this keyword pattern can indeed match the sentence 'Uni is fun', with `[phrase1?]` standing for nothing and `[phrase2?]` standing for 'is fun'. The same thing can be done also for the other terms, giving possibilities such as the following:

`[letter?]` Any single 'letter', or alternatively nothing at all.

`[!?]` Any punctuation mark or sequence of punctuation, but possibly nothing.

`[term?]` Any contiguous sequence of alphanumeric characters, or alternatively nothing at all.

`[phrase?]` Any sequence of one or more complete words consisting entirely of 'letters', but possibly nothing.

`[x?]` Any sequence of one or more complete 'items' and/or punctuation, but possibly nothing.

`[bracket?]` Like `[x?]`, but will not match any sequence that contains an 'unpaired' curved bracket.

Again abbreviation and variation is possible – the behaviour of the term is determined purely by the first character of the name ('l', ';', 't', 'p', 'x', or 'b' respectively in these cases) and whether the name ends with a question-mark.

3.1.6 Increasing or Decreasing Search

Most of the patterns above (apart from those that match only individual characters) are capable of matching text of various lengths. Hence it can make a difference whether the system searches through the possibilities in 'increasing' or 'decreasing' order – consider for example what happens when the pattern:

```
[word1]IN[word2]
```

is matched against the text word:

```
winning
```

This can be matched in two ways, either by matching `[word1]` with 'w' and `[word2]` with 'ning', or alternatively by matching `[word1]` with 'winn' and `[word2]` with 'g'. Which of these possibilities will be selected by *Elizabeth* will depend on which of them is found first in the search order. The search is done starting from the left, hence the order depends on what happens with the leftmost pattern, `[word1]` – the first alternative is the result of using an 'increasing' search for a match (i.e. a match of 'w' is tried, then 'wi', then 'win' etc.), and the second is the result of using a 'decreasing' search (i.e. a match of 'winning' is tried, then 'winnin', then 'winni' etc.).

The rule here is very simple: if the pattern begins with a *lower-case* letter (as is the case with `[word1]`), then the search will be done in *decreasing* order; if on the other hand the pattern begins with an *upper-case* letter (as is the case with `[Word1]`), then the search will be done in *increasing* order. Most of the time, decreasing order is appropriate, with the only frequent exception being `[X]` patterns. So in general it is advisable to start all other patterns with a lower-case letter, until for some reason it turns out that increasing order is what you need for the case in hand.

So in general, use lower-case patterns (e.g. `[letter]`, `[char]`, `[word]`, `[phrase]` rather than `[Letter]`, `[Char]` etc.), except in the case of `[X]` patterns, for which upper-case is usually preferable (e.g. `[X1]` and `[X2]` rather than `[x1]` and `[x2]`).

Note that the choice between increasing and decreasing search orders applies also with patterns containing a '?', e.g. `[word1?]` and `[letter2?]`, which can match *either* with a single word/letter *or* with nothing. Here again a pattern that begins with a lower-case letter will be tested in decreasing order (i.e. the non-null match will be tried before the null match), but upper-case can be used if you wish to search in increasing order. (Note that with the punctuation patterns `[,?]`, `[.?]`, `[;?]` and `[!?]`, the non-null match will always be tried before the null match, i.e. in decreasing search order. Here increasing order is unlikely ever to be needed, because punctuation characters only ever occur as 'singletons', with a space on either side (and the main practical use of `[!?]` is simply to ignore any sequence of punctuation, for example when directly followed by `[]` in testing whether some word or pattern is at the end of the active text).

For more detail on how pattern matching operates, see §5.3, 'Technical Note on Pattern Matching'. For an explanation of how these facilities can provide a means of implementing and experimenting with simple grammars, see §4.1, 'Implementing Grammatical Rules'.

3.2 Recursion and Text Splitting

One of the system's most powerful facilities is its capability for *recursion*, i.e. for feeding new inputs into itself. As an example, suppose that whenever you receive user input containing punctuation such as commas, semicolons, or question marks, you want this to be treated as though only the very first phrase had been typed. You can achieve this very simply by including the following as your first keyword and response:

```
K [phrase1] [;] [X1]
 R {[phrase1]}
```

The keyword will match any input that contains an initial phrase, followed by a punctuation mark and then by something else. The response is just the initial phrase, but the curly brackets around it indicate that instead of just being output as it stands (or after processing by output and final transformations if there are any), the phrase is to be fed back into the system as though it itself had been typed in as the user input.

3.2.1 Text Splitting and Recombining using Keyword Transformations

Another use of recursion is to enable the active text to be split into parts which are then to be processed separately before finally being recombined. Suppose, for example, that you want any text containing two or more single-phrase sentences (ending with full-stops, exclamation or question marks) to be split in this way. Try adding the following keyword and response lines to the Illustrative Script in §1.4:

```
19  K [phrase1] [.1] [phrase2] [.2]
20   R {[phrase1] [.1]} {[phrase2] [.2]}
```

and then type in two sentences that will each trigger an existing keyword, e.g.:

```
Mum is sad. I think she is ill.
TELL ME MORE ABOUT YOUR FAMILY. WHY DO YOU THINK SHE IS ILL?
```

Notice how this response is produced by lines 5, 19, 20, 26, 28, 24, and 25 (and again recall that you can see all this in action by clicking on the 'Trace' tab in *Elizabeth's* display):

(a) 'mum' is changed to 'mother' by the input transformation at line 5;

(b) in line 19, `[phrase1]` matches with 'mother is sad' and `[phrase2]` with 'i think she is ill' (with `[.1]` and `[.2]` matching the two full stops – the point of using two different patterns here is to allow for sentences that end with different punctuation);

(c) these substitutions are then inserted into the response at line 20, which (because of the curly brackets) feeds each of the two phrases back into *Elizabeth*, to be processed as though they were separate inputs before being ultimately recombined;

(d) the first of the phrases, 'mother is sad', matches the keyword MOTHER at line 26, and triggers the response 'TELL ME MORE ABOUT YOUR FAMILY' at line 28 (this is a random choice – 29 or 30 could equally well have been chosen instead);

(e) the second phrase, 'i think she is ill', matches the keyword 'I THINK `[phrase]`' at line 24, and when 'she is ill' is substituted back for `[phrase]` in the response at line 25, this gives 'WHY DO YOU THINK she is ill?';

(f) the two responses from steps (d) and (e) are recombined, hence after capitalization etc. the final output is: 'TELL ME MORE ABOUT YOUR FAMILY. WHY DO YOU THINK SHE IS ILL?'

3.2.2 No-Keyword Responses, Text Recombining, and Output/Final Transformations

Adapting the illustrative script as described above has a drawback when no keyword is recognised in a multiple-sentence input, for example:

```
Hello. Nice to meet you.
HELLO. NICE TO MEET ME.
```

The problem here is that the recursive keyword at line 19 is being used to split the input, so as far as *Elizabeth* is concerned, a keyword has indeed been recognised and hence there is no call for a no-keyword response (such as 'TELL ME WHAT YOU LIKE DOING'). But then since neither of the two split sentences ('Hello' and 'Nice to meet you') triggers any other keyword, they are simply echoed back before being recombined to generate the output 'HELLO NICE TO MEET ME.' (Note that this 'echoing' behaviour can be changed to 'blanking' – i.e. deletion – by the relevant setting in the Processing menu, which can be particularly useful when handling recursive text-splitting.)

The simplest way of dealing with this sort of problem is to replace the no-keyword response with a final keyword (and accompanying response) that will *always* match if it is ever reached, for example:

```
35  K [X?]
36   R TELL ME WHAT YOU LIKE DOING.
```

This will act much the same as the corresponding no-keyword response (which will now never be invoked), except in the case of split inputs:

```
Hello. Nice to meet you.
TELL ME WHAT YOU LIKE DOING. TELL ME WHAT YOU LIKE DOING.
```

Here each of the individual input sentences is being processed separately, but because neither of them matches any earlier keyword, the final keyword set is used in each case to generate the response 'TELL ME WHAT YOU LIKE DOING.', and these two responses are then recombined to produce the repetitive output.

Fortunately there is a way of dealing with this kind of duplication by using final transformations, which are just like output transformations except that they are always applied *after* any split text elements have been recombined. Repetition of output sentences can accordingly be eliminated by using:

```
37  F [X] [.] [X] [.] => [X] [.]
```

Which finally achieves the desired response:

```
Hello. Nice to meet you.
TELL ME WHAT YOU LIKE DOING.
```

3.2.3 Recursion and Punctuation

When handling text recursively, it is important to bear in mind how punctuation is treated. As explained in §2.3, 'The Input/Keyword/Output/Final Transformation Process', by default the first stage of processing the user input ensures that either a full stop, an exclamation mark or a question mark is present initially (unless the input is completely 'void'). From then on, no punctuation is added until just before the final output is displayed, where again, if the active text is not void, a full stop will be appended if necessary to ensure that the output ends with a complete sentence. The examples given above all take account of these features, ensuring that punctuation is recursed, where necessary, together with the text that precedes it. (For details of how final punctuation is dealt with in the matching of transformations, see §5.3, 'Technical Note on Pattern Matching'.)

Note however that all checking of final punctuation can be disabled by deselecting 'Check final punctuation' in the Processing menu. If you do this, then the system will never add any punctuation automatically.

3.2.4 Processing a List of Items, One at a Time

The example above enabled *Elizabeth* to deal with input involving two sentences, but suppose you wanted to extend this to handle a sequence of phrases however long that sequence might be. This can be done as follows:

```
K [phrase] [;] [X]
 R {[phrase]} {[X]}
```

As in our first example, the keyword will match any input that contains an initial phrase, followed by a punctuation mark and then by something else. But this time, as well as feeding back the initial phrase into *Elizabeth*, the response *also* feeds back the remainder of the input. This can then get processed in the same way, so that if it consists of a number of phrases, the first one will be picked out by the keyword and the remainder re-processed, and so on. This is a standard technique of 'list processing', whereby the first item of the list is handled separately while the rest of the list is recycled, and it is very commonly used in computer science and especially in artificial intelligence applications. If you find it at all hard to see what's going on here, try out the following simpler example:

```
K [term] [expression]
 R {[term]} {[expression]}
K 1
 R ONE
K 2
 R TWO
```

and then examine the trace after inputting something like '1 2 1 1 2 1'. This should enable you to see how the list of '1's and '2's is being broken down one at a time, with each individual '1' and '2' ultimately being replaced by 'ONE' and 'TWO' respectively. Note that `[term]` and `[expression]` here behave like `[word]` and `[phrase]` respectively, except that they allow the items to include digits as well as letters.

3.2.5 Recursing from Input/Output/Final Transformations

Recursion can be applied not only within keyword responses, but also input, output and final transformations, for example:

```
F [phrase1] [;] [phrase1] => {[phrase1] [;]}
```

This has the effect of identifying when a phrase is repeated (with punctuation between) at the stage of final transformation, and then sending back just a single copy for re-processing. Note that any part of the phrase not matched by the transformation (in this case, for example, the final punctuation mark) will go on to be processed as usual, and recombined with the recursive-processing result later (the best way to see this is to try it and look at the trace results). If this is not wanted, then it may be important to ensure that any such final transformation matches the entire text in any relevant case, for example:

```
F [] [X1?] [;1] [phrase1] [;2] [phrase1] [;2] [X2?] [] => {[phrase1]
[;2]}
```

This will match any text that includes two adjacent occurrences of the same phrase lying between punctuation marks, and ending with the same punctuation mark. One copy of the phrase plus punctuation mark is then sent back for re-processing, with no remainder, since the two occurrences of '`[]`' ensure that the match is with the entire string, while '`[X1?]`' and '`[X2?]`' are capable of 'eating up' any text that lies either side of the two occurrences of the identified phrase. Recursion is normally easier to manage with keywords, precisely because a response completely replaces the active text, rather than just a part of it. (For details of how the system handles this internally, see §5.3, 'Technical Note on Pattern Matching'.)

3.2.6 Targetted Recursion

By default, text re-processed through recursion is sent back to the beginning of the processing cycle, starting with the first input transformation. However this behaviour can be modified, so that re-processing of any chunk of text is 'targetted' to start from a particular point, for example:

```
Kthis [phrase] [;] [X]
 R {[phrase]=>O} {[X]=>Kthis}
```

sends '`[phrase]`' for recursion directly to the output transformations (bypassing input transformations and keywords), while sending '`[X]`' to the keyword transformations starting from the keyword set with index code 'this' (which happens to be this very keyword). Likewise '`=>Irepeat`' recurses back to the input transformation whose index code is 'repeat', and '`=>F003`' forward to the final transformation whose index code is '003' (thus bypassing input, keyword and output transformations, as well as any final transformations that precede F003). In all these cases, the index codes are alphabetically ordered (as shown in the tables), and processing starts from the first index code which is *equal to or alphabetically later* than the specified code.

3.2.7 Debugging: the 'Pause' Facility

Scripts that involve recursion, iteration, or cycling can sometimes be hard to understand, even with the help of the trace display, because so many operations can be performed so quickly. To help in 'debugging' such scripts, and finding exactly where they may be going wrong, the system provides a simple 'pause' facility, which can be invoked at any point in a script by including a dynamic command beginning with 'P' (see §4.2, 'Dynamic Script Processing'), for example:

```
K [term] [expression]
 R {[term]} {[expression]}
   & {!pause}
K 1
 R ONE
K 2
 R TWO
```

When the 'pause' command is executed, the user is asked whether to continue or not; if not, then the execution terminates exactly as though the *match algorithm limit* (explained in §3.3.1) had been exceeded, and this gives an opportunity to examine the state of processing at that point. Pause commands will generally need to be preceded by '!' to yield the desired effect, since this causes them to be enacted immediately (see §5.4,

'Sequencing and Timing of Dynamic Commands'). If you want to pause at a number of different points within your script, and to trace the sequence of this processing, then you can distinguish between the different pause locations by using different pause commands: whereas 'pause' or 'p' will show the standard message 'Pausing … Do you wish to continue?', any other pause command will itself be included in the pausing prompt, so for example 'Pause 1' will show the message 'Pause 1 … Do you wish to continue?'.

3.3 The Power of Recursion

Recursion is an extremely powerful technique; indeed theoretically, given its ability to recurse, most of the capabilities of *Elizabeth* could be generated by using *only* keywords and responses if that were desired, since for example an input transformation such as:

```
I dad => father
```

is essentially equivalent to the initial keyword/response pair:

```
K [X1?] dad [X2?]
 R {[X1?] father [X2?]}
```

Of course the latter is far less intuitive to understand, and can also introduce additional complications involving command sequencing etc., so it is generally far easier to use input transformations for this sort of simple substitution task. But it is worth bearing in mind the power and flexibility of recursion, which is available for almost any sort of repeated processing, and can be invoked from input/output/final transformations as well as from keywords.

As a classic illustration of the power of recursion, we can produce a general solution to the famous 'Towers of Hanoi' puzzle. This involves three pegs, numbered 1, 2, and 3, and several disks of increasing size – if there are five of these, we can call them A, B, C, D, and E. The disks have holes through their centres so they can fit onto the pegs, and initially all of the disks are piled up on peg 1, with the largest (disk E) at the bottom and the smallest (disk A) at the top of the pile. Our task is to give instructions for moving all of the disks from peg 1 onto peg 2, subject to the rules that *only one disk can be moved at any time, from one peg to another* and that *no disk can ever be placed on top of a smaller disk*. The key to this puzzle lies in the following insight:

> *Moving a pile of n disks from peg X to peg Y involves three operations: first, move the top (n-1) disks from peg X to peg Z; then move the n^{th} disk from peg X to peg Y; finally move the top (n-1) disks from peg Z to peg Y.*

Notice that this rule reduces the problem of moving *n* disks to the problem of moving *(n-1)* disks twice (plus one singleton move), after which the rule can be applied again to reduce each of these *(n-1)* moves to moves of *(n-2)* disks (etc.), and so on. Eventually, the problem is reduced all the way down to singleton moves, at which point it is completely solved. Here is a Script for achieving this, which is remarkably simple:

```
W GIVE A COMMAND SUCH AS 'MOVE ABCD FROM 1 TO 2'
I not12 => 3
I not21 => 3
I not13 => 2
I not31 => 2
I not23 => 1
I not32 => 1
K MOVE [word][letter] FROM [dig1] TO [dig2]
 R {MOVE [word] FROM [dig1] TO not[dig1][dig2]} ; MOVE [letter] FROM
    [dig1] TO [dig2] ; {MOVE [word] FROM not[dig1][dig2] TO [dig2]}
```

If, for example, you input 'MOVE ABC FROM 1 TO 2', you will get the following output:

```
MOVE A FROM 1 TO 2; MOVE B FROM 1 TO 3; MOVE A FROM 2 TO 3; MOVE C FROM
1 TO 2; MOVE A FROM 3 TO 1; MOVE B FROM 3 TO 2; MOVE A FROM 1 TO 2.
```

To see how this works, consider the first step, when the initial input is matched with the keyword, by substituting 'AB' for `[word]`, 'C' for `[letter]` *(note how this combination always ensures that `[letter]` matches the last disk in the pile, leaving `[word]` holding the rest)*, '1' for `[dig1]` and '2' for `[dig2]`. This generates the response:

```
{MOVE AB FROM 1 TO not12} ; MOVE C FROM 1 TO 2 ; {MOVE AB FROM not12 TO
2}
```

The middle instruction is now in place – to move disk C from peg 1 to peg 2 – but what is needed is to fill out the instructions on either side, to move the pile of disks AB first from peg 1 to peg 3, then finally from peg 3 to peg 2. Note how 'not12' here is used to refer to peg 3 (the peg which is *not* peg 1 or peg 2): this code is dealt with by the six input transformations, which on each recursion ensure that an appropriate replacement is made. And that's it! Explaining how the first step of this process works implicitly explains them all, because the filling out of the remaining instructions takes place in exactly the same sort of way, using the same single rule. So we have here a general solution to the Towers of Hanoi, which can in principle deal with the movement of any number of disks from any peg to any other (try 'MOVE ABCDEF FROM 3 TO 2', for example). This Script is available in the file `Hanoi.txt`.

3.3.1 Recursion, Iteration, Cycling, and Overload

By default, recursion is permitted within *Elizabeth*, but it can be switched off by cancelling the appropriate checkbox in the Control menu dialog. One reason for doing this would be to take advantage of the automatic iteration and cycling control facilities explained in §5.2, 'Iteration and Cycling of Input/Output/Final Transformations', since these cannot be combined with recursion. Note that, just like iteration and cycling, recursion is a potential cause of infinite looping. To avoid this, the Control menu dialog provides the 'backstop' of an adjustable *match algorithm limit* which by default is set to 5,000 (there is also a related *memory checking limit*, by default 500 and set through the same Control menu dialog – see §6.1.4 for the explanation of this). This means that further processing will not take place after the number of calls to the matching algorithm that drive the various transformations exceeds 5,000. So if you get unexpected results from a recursive script (e.g. if the output contains curly brackets), the first thing to check is whether this sort of overload might be the cause. If you cannot remove the possibility of overload from your Script, note that the 'H' script command provides a facility for generating a specific message when the system halts for this reason – see the arithmetical scripts in the following section for illustrations of this.

3.3.2 Using Recursion and Cycling for Arithmetic

Another illustration of the power of recursion is the general implementation of arithmetical capabilities based on *Elizabeth's* simple functions INC and DEC (explained in §6.2, 'Built-In String Functions'). INC and DEC respectively 'increment' and 'decrement' a string – where that string represents a number, these operations are defined so as to correspond respectively to arithmetical addition of 1, and subtraction of 1. To see how general addition, subtraction and multiplication can be defined recursively based on these two functions, load the following Script and then type in some questions like 'What is -3 plus 6?', 'What is 9 minus 5?', 'What is -4 times 3?' and so on (but note that multiplication can take a very long time, especially when the second number is large – 'overload' is dealt with by the 'H' command which is explained in the previous section and used below).

```
W TYPE IN A QUESTION LIKE 'WHAT IS -4 PLUS 7?' - USING 'PLUS', 'MINUS',
OR 'TIMES'.

I --[number] => [number]

K WHAT IS [term1] PLUS [term2]
 R THE SUM IS {ADD([term1],[term2])}

K WHAT IS [term1] MINUS [term2]
 R THE DIFFERENCE IS {ADD([term1],-[term2])}

K WHAT IS [term1] TIMES [term2]
 R THE PRODUCT IS {MULT([term1],[term2])}

K [X1?] ADD([term],0) [X2?]
 R {[X1?] [term] [X2?]}

K [X1?] ADD([term],-[number]) [X2?]
 R {[X1?] ADD([dec:[term]],[inc:-[number]]) [X2?]}

K [X1?] ADD([term],[number]) [X2?]
 R {[X1?] ADD([inc:[term]],[dec:[number]]) [X2?]}
```

```
K [X1?] MULT([term],0) [X2?]
 R {[X1?] 0 [X2?]}

K [X1?] MULT(-[number1],-[number2]) [X2?]
 R {[X1?] ADD([number1],MULT(-[number1],[inc:-[number2]])) [X2?]}

K [X1?] MULT([number1],-[number2]) [X2?]
 R {[X1?] ADD(-[number1],MULT([number1],[inc:-[number2]])) [X2?]}

K [X1?] MULT([term],[number]) [X2?]
 R {[X1?] ADD([term],MULT([term],[dec:[number]])) [X2?]}

H SORRY - THAT'S TOO HARD FOR ME!
```

Suppose, for example, that you input ‘What is 6 plus 4?’ The first keyword transformation will replace this with ‘THE SUM IS’ plus the result of treating ‘ADD(6,4)’ recursively. ‘ADD(6,4)’ will be dealt with repeatedly by the sixth keyword transformations, being replaced first by ‘ADD(7,3)’ (because 7 is `[inc:6]` and 3 is `[dec:4]`), followed by ‘ADD(8,2)’, ‘ADD(9,1)’, and ‘ADD(10,0)’, at which point the fourth keyword transformation will deliver 10 as the result. Multiplication is, of course, more complicated, but the basic principle is straightforward enough: the final keyword transformation, for example, replaces ‘MULT(4,7)’ with ‘ADD(4,MULT(4,6))’ – the initial multiplication is reduced to an addition involving a *smaller* multiplication, and this in turn is reduced to another addition involving a *yet smaller* multiplication, and so on. Eventually the second term of the multiplication reaches 0, at which point the calculation has been reduced entirely to additions.

In practice, this sort of recursive processing is easier to handle in *Elizabeth* using cycling input transformations rather than recursive keyword transformations, as follows:

```
W TYPE IN A QUESTION LIKE 'WHAT IS -4 PLUS 7?' - USING 'PLUS', 'MINUS',
OR 'TIMES'.

I --[number] => [number]

K WHAT IS [term1] PLUS [term2]
 R THE SUM IS {ADD([term1],[term2])}

K WHAT IS [term1] MINUS [term2]
 R THE DIFFERENCE IS {ADD([term1],-[term2])}

K WHAT IS [term1] TIMES [term2]
 R THE PRODUCT IS {MULT([term1],[term2])}

I ADD([term],-[number]) => ADD([dec:[term]],[inc:-[number]])
I ADD([term],0) => [term]
I ADD([term],[number]) => ADD([inc:[term]],[dec:[number]])

I MULT(-[number1],-[number2]) => ADD([number1],MULT(-[number1],[inc:-
[number2]]))
I MULT([number1],-[number2]) => ADD(-[number1],MULT([number1],[inc:-
[number2]]))
I MULT([term],0) => 0
I MULT([term],[number]) => ADD([term],MULT([term],[dec:[number]]))

H SORRY - THAT'S TOO HARD FOR ME!
```

The principles are the same, but the Script is smaller and simpler, because input transformations are designed to cycle and to handle parts of strings, and so don’t need explicit recursion or the `[X1?]` and `[X2?]` patterns.

4. Advanced Script Processing

4.1 Implementing Grammatical Rules

The system has been designed so that the same mechanisms which enable it to function as a traditional ELIZA 'chatbot' are sufficiently abstract and general to be easily adaptable for other purposes, and in particular, to provide an introduction to some of the major concepts and techniques of natural language processing (NLP). Prominent among these is the implementation of grammatical rules. (Another more advanced application of these mechanisms is illustrated in §4.6 on 'Implementing Propositional Logic')

4.1.1 Grammatical Analysis of Sentences

If any of the material described here appears not to work as you expect, consult the 'Technical Notes' at the bottom of this section for an explanation.

As a first example, consider the following simple grammar which specifies rules for the composition of sentences, noun phrases, and verb phrases respectively:

S => NP VP

NP => D N

VP => V NP

with words assigned to categories as follows:

D (determiner): a, the

N (noun): cat, dog, rabbit

V (verb): bites, chases, likes

These rules and assignments enable us to analyse simple sentences such as the following:

The cat bites the dog

A rabbit likes the cat

The dog chases a rabbit

In order to do this within the *Elizabeth* system, we need to translate both the grammatical rules and the category assignments into appropriate transformations, and then display back the finished result. We can do this as follows (and see §3.1 on 'Pattern Matching' for details of the terms involved, in particular the `[brak]` terms which importantly check for matching brackets):

```
I a => (d: A)
I the => (d: THE)
I cat => (n: CAT)
I dog => (n: DOG)
I rabbit => (n: RABBIT)
I bites => (v: BITES)
I chases => (v: CHASES)
I likes => (v: LIKES)
I (d: [brak1]) (n: [brak2]) => (np: (D: [brak1]) (N: [brak2]))
I (v: [brak1]) (np: [brak2]) => (vp: (V: [brak1]) (NP: [brak2]))
I (np: [brak1]) (vp: [brak2]) => (s: (NP: [brak1]) (VP: [brak2]))
W TYPE A SENTENCE USING:  A, THE, CAT, DOG, RABBIT, BITES, CHASES, LIKES
```

Here the input transformations are designed so that the user input sentence will be transformed in the following sort of way, with the bracketing reflecting the structure of the analysis as it is progressively built up:

```
the cat likes a dog

(d: THE) (n: CAT) (v: LIKES) (d: A) (n: DOG)

(np: (D: THE) (N: CAT)) (v: LIKES) (np: (D: A) (N: DOG))
```

```
(np: (D: THE) (N: CAT)) (vp: (V: LIKES) (NP: (D: A) (N: DOG)))
(s: (NP: (D: THE) (N: CAT)) (VP: (V: LIKES) (NP: (D: A) (N: DOG))))
```

Notice how each word or grammatical category code is introduced first in lower case, but then changed to upper case after it has been absorbed within a larger category. This change ensures that the word or code is not re-used by another transformation, all of which are specified in terms of lower-case words or codes on their left-hand side. (This use of capitalisation is similar in principle to that explained in §5.1, 'Capitalisation and Transformations'; see also §6.2, 'Built-In String Functions', which introduces the functions `[UPPER:]` and `[LOWER:]`, enabling you to control the use of upper and lower case in the various transformations)

Once all the input transformations have been completed the system would normally move on to process keyword transformations and no-keyword responses, but since the script hasn't defined any of these, it simply gives as its response whatever the active text may be at that point (as explained in §2.2, 'Simple Message Selection Commands'). To see all this in operation, just copy the Script lines shown above into a text file, load it into *Elizabeth*, and type 'The cat likes a dog' in response to the welcome message 'TYPE A SENTENCE USING: A, THE, CAT, DOG, RABBIT, BITES, CHASES, LIKES'. (Alternatively, load the script file `Passive.txt` – which incorporates also the commands in the next section – and examine the Trace display to see the intermediate stages of processing.)

4.1.2 Simple Grammatical Transformations

Once a sentence has been analysed grammatically, it becomes much easier to transform it 'intelligently' in various ways. A very simple application on the grammar above is to transform the sentences from active to passive, so that for example the input:

```
a rabbit chases the cat
```

is transformed into the output:

```
THE CAT IS CHASED BY A RABBIT
```

This can be achieved by adding to the Script above the following keyword and output transformations (and note here that '`[b1]`', '`[b2]`' etc. can serve as well as '`[brak1]`', '`[brak2]`' etc. for bracket-matched terms):

```
K (s: (NP: [b1]) (VP: [b2]))
 R (s: (VP: [b2] passive) (NP: [b1]))

O (s: (VP: [b1] passive) (NP: [b2])) => (vp: [b1] passive) (np: [b2])
O (vp: (V: [b1]) (NP: [b2]) passive) => (np: [b2]) (v: [b1] passive)
O (np: (D: [b1]) (N: [b2])) => (d: [b1]) (n: [b2])
O (v: BITES passive) => IS BITTEN BY
O (v: CHASES passive) => IS CHASED BY
O (v: LIKES passive) => IS LIKED BY
O (d: [b1]) => [b1]
O (n: [b1]) => [b1]
```

The keyword transformation changes the active sentence structure of a noun phrase followed by a verb phrase to a passive sentence structure, consisting of a corresponding passive verb phrase followed by the noun phrase. Then the first three output transformations decompose this structure into its constituents, the next three substitute the appropriate passive verb form, and the last two unpack the determiners and nouns. This entire example is provided in the script file `Passive.txt`.

4.1.3 More Complicated Transformations

More sophisticated sentence transformations usually require 'agreement' between the various parts of the sentence, and an excellent example to illustrate this is provided by so-called 'tag questions'. A tag question is appended to the end of a sentence, to ask for confirmation or to give emphasis to what was said, for instance:

'John chases the cat.'
becomes 'John chases the cat, doesn't he?'

'The white rabbits bit the black dog.'
becomes 'The white rabbits bit the black dog, didn't they?'

'You like her'
becomes 'You like her, don't you?'

Notice how the tag question must agree with the sentence in all of *number* (singular or plural), *person* (first person, second person, third person), *gender* (masculine, feminine, neuter), and *tense* (past, present, future).

To implement the tag question transformation in *Elizabeth*, therefore, it is necessary to record the number, person, gender, and tense of all the relevant words, and then to select the appropriate question accordingly. This can be achieved using the following style of Script, in which `[bw]` is typically used to match words and their basic categories (noun, verb etc.) and phrase structures, `[wn]` matches the *number* ('sing' or 'plur'), `[tp]` matches the *person* ('1st', '2nd', '3rd'), `[wg]` matches the *gender* ('masc', 'fem', 'neut'), and `[wt]` matches the *tense* ('pres', 'past'). Note that a `[b…]` field *must* be used for the words and categories, because these involve complex terms with internal bracketed structure etc.; but it's best to use `[w…]` or `[t…]` fields for the other terms, because we know that these all involve a single word or term (a *word* just contains letters, whereas a *term* can include digits also).

```
W TYPE A SENTENCE USING: A, THE, MAN, MEN, WOMAN, WOMEN, DOG, DOGS, I,
WE, YOU, HE, SHE, IT, THEY, LIKES, LIKE, LIKED, CHASES, CHASE, CHASED.

I a => (d: A)
I the => (d: THE)
I man => (n: MAN,sing,3rd,masc)
I men => (n: MEN,plur,3rd,masc)
I woman => (n: WOMAN,sing,3rd,fem)
I women => (n: WOMEN,plur,3rd,fem)
I dog => (n: DOG,sing,3rd,neut)
I dogs => (n: DOGS,plur,3rd,neut)
I i => (p: I,sing,1st,any)
I we => (p: WE,plur,1st,any)
I you => (p: YOU,sing,2nd,any)
I he => (p: HE,sing,3rd,masc)
I she => (p: SHE,sing,3rd,fem)
I it => (p: IT,sing,3rd,neut)
I they => (p: THEY,plur,3rd,any)
I likes => (v: LIKES,pres)
I like => (v: LIKE,pres)
I liked => (v: LIKED,past)
I chases => (v: CHASES,pres)
I chase => (v: CHASE,pres)
I chased => (v: CHASED,past)

I (p: [bw],[wn],[tp],[wg]) => (np: (P: [bw]),[wn],[tp],[wg])
I (d: [bw1]) (n: [bw2],[wn],[tp],[wg]) => (np: (D: [bw1]) (N:
[bw2]),[wn],[tp],[wg])
I (v: [bw1],[wt]) (np: [bw2],[wn],[tp],[wg]) => (vp: (V: [bw1]) (NP:
[bw2]),[wt])
I (np: [bw1],[wn],[tp],[wg]) (vp: [bw2],[wt]) => (s: (NP: [bw1]) (VP:
[bw2]),[wn],[tp],[wg],[wt])

K (s: [bw],[wn],[tp],[wg],[wt])
R (s: [bw]) , (t: [wn],[tp],[wg],[wt]) ?

O (s: (NP: [bw1]) (VP: [bw2])) => (np: [bw1]) (vp: [bw2])
O (vp: (V: [bw1]) (NP: [bw2])) => (v: [bw1]) (np: [bw2])
O (np: (D: [bw1]) (N: [bw2])) => (d: [bw1]) (n: [bw2])
O (np: (P: [bw])) => (p: [bw])

O (d: [w]) => [w]
```

```
O (n: [w]) => [w]
O (p: [w]) => [w]
O (v: [w]) => [w]

O (t: [wn],[tp],[wg],[wt]) => (do: [wn],[tp],[wt]) (pro: [wn],[tp],[wg])
O (do: [wn],1st,pres) => DON'T
O (do: [wn],2nd,pres) => DON'T
O (do: sing,3rd,pres) => DOESN'T
O (do: plur,3rd,pres) => DON'T
O (do: [wn],[tp],past) => DIDN'T
O (pro: sing,1st,[wg]) => I
O (pro: plur,1st,[wg]) => WE
O (pro: [wn],2nd,[wg]) => YOU
O (pro: sing,3rd,masc) => HE
O (pro: sing,3rd,fem) => SHE
O (pro: sing,3rd,neut) => IT
O (pro: plur,3rd,[wg]) => THEY
```

This Script is available in the file `TagQuest.txt`.

4.1.4 An Interesting Exercise

There is obviously plenty more that can be done in the way of experimenting with grammatical transformations. One fairly straightforward but entertaining possibility is to create a script that generates arbitrarily long sentences of the 'house that Jack built' style, by using relative pronouns such as 'who', 'which', and 'that' to string together noun and verb phrases. Using a development of the grammar above, this might produce 'The man who chased a dog that likes the dog that chased the woman likes her.' etc.

4.1.5 Technical Notes

1. In the discussion above, grammatical rules and transformations have been shown with optimal spacing for reading (so for example '(N: DOG)' contains only one space). However within Elizabeth, spaces are automatically inserted in such a way as to simplify processing, so that you don't need to worry about spacing when defining your rules; hence standard spacing is applied only to the final output.

2. Grammatical rules are characteristically recursive, with structures occurring within other structures. Hence their implementation within the system will often require, at least in complex cases, either the explicit use of recursion, or else (if implemented in the way illustrated above) that both input and output transformation sets can be repeatedly 'cycled' – to permit this, choose the appropriate settings from the Control menu dialog, as explained in §5.2, 'Iteration and Cycling of Input/Output/Final Transformations'. Note that the predefined scripts mentioned above (and available through the Help menu) automatically apply these settings, using the '/C' directive which is also explained in that section.

4.2 Dynamic Script Processing

The section 'Phrase Memorisation and Recall' (§2.4) contains an illustration of dynamic processing. There it is shown how the following command:

```
K [] my [phrase]
  & {M [phrase]}
```

not only defines a keyword, but also specifies an action to be performed when that keyword is 'matched' – namely, that a corresponding phrase should be memorised using the 'M' command. This is an example of 'dynamic processing' because it involves a dynamic modification to the Script while the conversation is in progress.

4.2.1 Commands Available for Dynamic Script Processing

This sort of dynamic processing is not confined to the memorisation of phrases; indeed any of the following types of script command can be enclosed within the curly brackets following '&' to specify an action:

W a 'welcome' message

Q a 'quitting' message

X an 'exit' message

V a 'void input' response

N a 'no-keyword' response

H an overload halting message

I an input transformation

K a keyword pattern

R a keyword response pattern

O an output transformation

F a final transformation

M a memorised phrase

P a 'pause', which need not affect the processing at all, but which offers the user an opportunity to interrupt any further pattern matching (for an explanation and example, see §3.2.7 on 'Debugging')

Moreover such commands can be 'triggered' by any of the following 'events':

use of a particular 'welcome' message (W)

use of a particular 'quitting' message (Q)

use of a particular 'exit' message (X)

use of a particular 'void input' message (V)

use of a particular 'no-keyword' message (N)

halting of processing on exceeding the *match algorithm limit* or *memory checking limit* (H – however note that an action can be associated with such halting even if the overload halting message is null and therefore not used)

successful application of an input transformation (I)

matching of a keyword, together with choice of an appropriate response phrase (K – note that no 'keyword event' occurs unless there is an appropriate response phrase to select)

use of a particular keyword response (R)

successful application of an output transformation (O)

successful application of a final transformation (F)

use of a particular memorised phrase (M)

A single event can be associated with a number of actions, and these themselves can create further event-action linkages. A relatively simple example might be as follows:

```
I my sister => my sister
  & {K MOTHER
       & {N DOES ANYTHING ELSE ABOUT YOUR MOTHER COME TO MIND?}
      \R HOW WELL DO YOUR MOTHER AND SISTER GET ON?}
```

This has the effect that when the phrase 'my sister' is identified within the user input, this is left unchanged (because it is replaced directly with 'my sister') but at that point a keyword and corresponding response are added to the Script, as though they had been included in the original Script with the following commands:

```
K MOTHER
  & {N DOES ANYTHING ELSE ABOUT YOUR MOTHER COME TO MIND?}
```

```
\R HOW WELL DO YOUR MOTHER AND SISTER GET ON?
```

From now on, 'MOTHER' will function as a keyword, with 'HOW WELL DO YOUR MOTHER AND SISTER GET ON?' as a corresponding response (which however can only be used once, since '`\R`' signifies a *self-deleting* response – see §4.2.4, 'Self-Deletion', for an explanation). Moreover if that keyword is matched and the response generated, the 'N' command in the curly brackets will then be triggered, creating at that point the no-keyword response 'DOES ANYTHING ELSE ABOUT YOUR MOTHER COME TO MIND?'. Since you can have brackets within brackets within brackets to as deep a level as you want, the possibilities here are limited only by your ingenuity and imagination!

4.2.2 Interaction of Dynamic Commands with the Existing Script

In the example just given, 'MOTHER' is defined as a keyword with a particular response and action. But what happens if (as in the Illustrative Script of §1.4) 'MOTHER' has already been defined as a keyword? The answer is that in this case, the existing 'MOTHER' keyword specification (including any associated actions) will simply be replaced by the new 'MOTHER' keyword specification, and the newly-defined response ('HOW WELL DO YOUR MOTHER AND SISTER GET ON?') will be added to the existing set of responses associated with that keyword. So before adding a new keyword (or void input response etc., or input/output/final transformation, or memorised phrase), the system searches through to see if that keyword (or void/input response etc., or memorised phrase, or an input/output/final transformation with the same left-hand side) already exists in the system. If it does already exist, then it is replaced (which will often mean no change at all, but could mean that its behaviour changes if a new action is added, or an old one removed). This automatic replacement means that if the same action is repeated numerous times (e.g. every time the 'MOTHER' keyword is matched), you don't end up with lots of copies of the same commands being added over and over again to the Script.

If you add a completely new keyword to the Script, then it will usually be added either to the last-modified keyword set, or (if that last-modified set already contains both at least one keyword and at least one response), to a new keyword set. A new keyword *response* will usually be added to the last-modified keyword set, but if no such set exists (either because no keywords have yet been defined, or because the last modification involved the entire deletion of a keyword set as explained in the next section), then it will be added to a newly created keyword set.

Full details of all these matters are provided in the Command Syntax Reference Guide (§6.1). Note that the interaction of dynamic commands with the existing Script can be controlled much more finely by making use of the 'index codes' attached to Script commands. For details and further examples of dynamic commands, see §4.3, 'Control of Scripts using Command Index Codes'. For explanation of *when* in the processing cycle dynamic commands are carried out (and how this can be controlled), see §5.4, 'Sequencing and Timing of Dynamic Commands'.

4.2.3 Deletion of Script Commands

All of the types of action listed above have a deletion variant, which is best explained by illustration:

```
V\ WOULD YOU LIKE TO TALK ABOUT SOMETHING ELSE?
```

deletes the void input response 'WOULD YOU LIKE TO TALK ABOUT SOMETHING ELSE?' *(together, of course, with any associated actions – however if no such void input response is currently defined in the system, then nothing happens);*

```
V\
```

deletes all current void input responses – in which case the program will respond to a void input with 'I CAN'T THINK OF ANYTHING TO SAY.' *(etc. – the italicised bracket in the previous clause applies to all of these examples);*

```
I\ dad => father
```

deletes the specific input transformation which replaces 'dad' with 'father' *(etc.);*

```
I\ dad
```

deletes the input transformation whose left-hand side is ‘dad’ *(etc. – note here that any specific deletion command will delete at most one item – so if there is more than one input transformation whose left-hand side is ‘dad’, then only the first (in index order) will be deleted;*

```
K\ MOTHER
```

deletes the keyword ‘MOTHER’ *(etc.);*

```
K\
```

deletes every keyword in every keyword set, but leaves the sets in existence *(etc.);*

```
K/\
```

deletes every keyword set, including all keywords and all responses *(etc.);*

```
R\ HOW WELL DO YOUR MOTHER AND SISTER GET ON?
```

deletes this particular response if it exists in the most recently modified keyword set *(etc.);*

```
R@\
```

deletes all responses in the most recently modified keyword set *(etc.);*

```
R\
```

deletes every response in every response set *(etc.);*

```
M\
```

deletes any phrases that are currently memorised (etc.);

```
M\ MOTHER [phrase]
```

deletes any currently memorised phrase that matches the pattern ‘MOTHER `[phrase]`’ (etc.). Note in this last example that *such use of patterns within deletions is allowed only with memorised phrases* (because memorised phrases cannot themselves contain bracketed patterns, so there is no risk of ambiguity) – in other cases (e.g. with ‘welcome’ messages etc., input/output/final transformations, keywords and keyword responses), such as:

```
K\ [phrase1] IS BETTER THAN [phrase2]
```

the patterns do not act as ‘wildcards’ – here the deletion applies just to the literal keyphrase ‘`[phrase1] IS BETTER THAN [phrase2]`’, exactly as shown.

Just as with command interaction, it is possible to control command deletion more finely by making use of the ‘index codes’ attached to Script commands. For details, see §4.3, ‘Control of Scripts using Command Index Codes’, and §6.1, ‘Command Syntax Reference Guide’.

4.2.4 Self-Deletion

The most common situation in which it is desirable to delete a message, transformation, keyword or response is when it has just been used and repetition would seem conversationally very unnatural. For this particular situation, *Elizabeth* provides a simple facility whereby any of these items can be defined as *self-deleting* (i.e. they ‘delete themselves’ after being used once). This is achieved by preceding the relevant command with ‘\’, for example:

```
\N CHANGING THE SUBJECT, DO YOU LIKE SPORT?
```

This illustration is from §2.2, ‘Simple Message Selection Commands’ – clearly the message here is not one that would naturally be repeated in a single conversation, and defining it as self-deleting removes any such risk. The same can be done with welcome, quitting, exit, and void-input messages; with input, output, and final transformations; and with keywords and responses. Here is an example involving keyword responses:

```
K MOTHER
 \R WOULD YOU SAY YOUR MOTHER IS CONTENTED?
 \R DID YOUR MOTHER HAVE A HAPPY CHILDHOOD?
 \R WAS YOUR MOTHER KIND TO YOU AS A CHILD?
```

```
K MOTHER
 R LET'S TALK ABOUT SOMETHING ELSE NOW
 \R WE'VE TALKED ENOUGH ABOUT YOUR MOTHER
```

Since there are two sets with exactly the same keyword, the second set will never be used while the first one is available. The first contains three responses, but all of them are self-deleting, so each can be used just once, providing three relevant responses altogether. If 'MOTHER' occurs a fourth time, the first set will be empty of responses and therefore unusable, so this keyword will instead be picked up by the second set, randomly choosing one of the two responses and then, on the fifth occasion, the other of them (note that one of the two is self-deleting). By that time only one response will be left: 'LET'S TALK ABOUT SOMETHING ELSE NOW', but this will never be deleted, so the system will continue to recognise the keyword 'MOTHER' and respond accordingly.

4.2.5 Using Memories in Dynamic Commands

Suppose that we have a simple Script which is designed to chat about things the user likes:

```
W TELL ME SOMETHING YOU LIKE
N TELL ME SOMETHING ELSE YOU LIKE

K I LIKE [word]
R OK, LET'S TALK ABOUT [word]
  & {Mtopic [word]
     <[Mtopic]!=[word]>: N SHALL WE GO BACK TO TALKING ABOUT [word]?}

K I THINK [phrase]
R WHY DO YOU THINK [phrase]?
```

which might give rise to the following dialogue:

```
TELL ME SOMETHING YOU LIKE
```
(the Welcome message)

```
I like dogs.
OK, LET'S TALK ABOUT DOGS.
```
(from the first keyword set, also saving 'dogs' as the value of `[Mtopic]`, and creating the no-keyword response:
`<[Mtopic]!=dogs>: N SHALL WE GO BACK TO TALKING ABOUT dogs?`
which is designed to give an impression of conversational memory, with the condition that the topic at the time should itself be different from 'dogs')

```
I think dogs are fun.
WHY DO YOU THINK DOGS ARE FUN?
```
(from the second keyword set)

```
They enjoy playing.
TELL ME SOMETHING ELSE YOU LIKE.
```
(the only no-keyword response available, because `[Mtopic]` currently has the value 'dogs', so the condition of the newly-created response 'SHALL WE GO BACK TO TALKING ABOUT dogs?' is not satisfied)

```
I like cats.
OK, LET'S TALK ABOUT CATS.
```
(from the first keyword set, also saving 'cats' as the value of `[Mtopic]`, and creating the no-keyword response:
`<dogs!=cats>: N SHALL WE GO BACK TO TALKING ABOUT cats?`
because when the creation command is generated, `[Mtopic]` has the value 'dogs')

```
Cats are very independent.
SHALL WE GO BACK TO TALKING ABOUT CATS?
```

Here something has clearly gone wrong, and the reason is fairly easy to see. At this point in the conversation, we have three no-keyword responses available, as defined by the following commands:

```
N TELL ME SOMETHING ELSE YOU LIKE
<[Mtopic]!=dogs>: N SHALL WE GO BACK TO TALKING ABOUT dogs?
<dogs!=cats>: N SHALL WE GO BACK TO TALKING ABOUT cats?
```

The first of these was defined in the original Script, the second was created after the first user input (at which point `[Mtopic]` was undefined), and the third was created later, at a point when `[Mtopic]` had been given the value 'dogs'. But because of this, the third no-keyword response has not come out as the intended:

```
<[Mtopic]!=cats>: N SHALL WE GO BACK TO TALKING ABOUT cats?
```

which would correctly check that the current topic is something other than 'cats' before becoming available. So the problem we have is this: if some memory variable currently has a value, then how can we tell *Elizabeth* when we want *the memory variable itself* to be inserted in a command, rather than *the current value of that memory variable*?

The solution to this problem is very simple, and generally applicable (e.g. to memorised inputs and outputs such as `[I-2]` and `[O-1]` as explained in §2.4.3, 'Automatic Memory of the Preceding Dialogue'. If you don't want a memory variable to be substituted by its current value within some command, then just insert an apostrophe as follows:

```
R OK, LET'S TALK ABOUT [word]
  & {Mtopic [word]
     <['Mtopic]!=[word]>: N SHALL WE GO BACK TO TALKING ABOUT [word]?}
```

Elizabeth applies the straightforward rule that any `['` sequence in a dynamic command is replaced by `[` *without making any further substitution*, so the response shown here will have the desired effect, when `[word]` stands for 'cats', of creating the no-keyword response:

```
<[Mtopic]!=cats>: N SHALL WE GO BACK TO TALKING ABOUT cats?
```

and this will happen *irrespective of the current value of `[Mtopic]`*.

Note finally that if you use commands within other dynamic commands, then you might occasionally have use for constructions such as `['']Mtopic]`, which reduces to `[Mtopic]` after two command applications rather than just one. Likewise `['']']Mtopic]`, `['']']']Mtopic]` etc. are also possible, though increasingly improbable.

4.3 Control of Scripts using Command Index Codes

Whenever a Script command is stored in the system, it is assigned an 'index code' which can then be used to identify it within further control commands (e.g. replacements or deletions). By default, these index codes take the form of three-digit numbers starting from '001' and counting up to '999' (beyond this, they go to 'A000', 'A001' etc.). But if you plan to do any dynamic scripting as introduced in §4.2, 'Dynamic Script Processing', then you will probably prefer to specify your own index codes, so as to be able to cross-refer from one command to another (e.g. when replacing or deleting an existing command, or ensuring that they are ordered in the way that you choose). Even more crucially, by using index codes for memorised phrases you will be able to store and recall of as many of them as you wish, rather than being confined to recalling only the latest few of them using the codes '`[M]`', '`[M-1]`', '`[M-2]`' or the first few of them using '`[M+1]`', '`[M+2]`' etc. as explained in §2.4, 'Phrase Memorisation and Recall'. And as explained in §4.4, 'Giving Direction to a Conversation', this also allows the use of stored phrases to control the conversational flow, extending the system's potential far beyond the bounds of the traditional ELIZA paradigm.

Note that with input, output, final, and keyword transformations, and also memorised phrases, the particular index codes that you choose have a direct impact on the system's behaviour, because they determine the order in which the various transformations are processed (i.e. in alphabetical order of index codes) or in which the memorised phrases are recalled (e.g. by treating the alphabetically last index code as the 'most recent'). For this reason it is often a good idea to use two-part codes such as `AA_family`, where the part before the underscore has the purpose of determining the ordering (here 'AA' would presumably be first in sequence) while the part after the underscore is descriptive. However in the examples that follow, for simplicity the index codes given are purely descriptive.

4.3.1 Assigning and Using Index Codes

In general, the assigning of index codes to commands is extremely straightforward – the relevant code simply needs to be inserted immediately after the command letter (and hence before the deletion backslash '\' if any). For example:

```
Idad dad => father
```

stores the relevant input transformation under the index code 'dad', so if you then wanted to replace it dynamically with an alternative transformation, you could do so by including in some relevant action the command:

```
Idad dad => parent
```

or delete it entirely with:

```
Idad\
```

Exactly the same principles apply to output and final transformations, and to the various 'fixed' messages such as void input and no-keyword responses. Thus

```
Vvoid1 DON'T BE SHY - TELL ME WHAT YOU ARE THINKING.
```

defines a void input response, which can later be replaced by using the command:

```
Vvoid1 YOU'RE STILL BEING SHY - PLEASE SHARE YOUR THOUGHTS!
```

As a slightly artificial illustration of the dynamic possibilities, you could include in your Script the following complex command:

```
Vv1 PLEASE SPEAK!
 & {Vv1 FOR THE SECOND TIME, PLEASE SPEAK!
     & {Vv1 FOR THE THIRD TIME, PLEASE SPEAK!
         & {Vv1 FOR THE LAST TIME, PLEASE SPEAK!
             & {Vv1\}}}}
```

Here the Script command creates the void input response 'PLEASE SPEAK!', but also defines an action to take place when this response is invoked (for explanation of such actions, see §4.2, 'Dynamic Script Processing'). That action, however, has the effect of replacing the original void input reponse – this is because both of them have the same index code of 'v1'. But again this replacement response ('FOR THE SECOND TIME …') itself has a similar type of associated action, so it in turn is replaced once used. Such replacement can happen yet again, but after a fourth void input response has been used, it is finally deleted, leaving no such responses at all in the Script. To see all this in operation, simply copy this command into your own Script, then load it and repeatedly click on the 'Enter' button while viewing the 'Void' table in the middle of the *Elizabeth* screen (if you can't see the 'Void/No-key' tab to click on, select 'View analysis tables' from the Analysis menu). Note also that whenever an action is associated with some command, the details can be viewed by clicking on the '[Click]' text in the last column of the relevant table row.

4.3.2 Assigning Index Codes to Keywords and Responses

Keywords and keyword responses are slightly more complicated, each having two index codes, one for the keyword set and one for the keyword or response itself. (To view how this works, load the default Script from the file `Elizabeth.txt` or `EOriginal.txt`, then look at the 'Keyword transforms' table in the bottom half of the *Elizabeth* screen – you'll see that the keywords and responses are grouped together in sets, with the set index being shown in the first column of the table and the individual keyword or response code in the third column.) To assign both index codes together, combine them with a slash '/' as follows:

```
Kfamily/mother MOTHER

Rfamily/more TELL ME MORE ABOUT YOUR FAMILY
```

These two commands (which need not be contiguously located within a Script) create a keyword and response within the same keyword set, whose index code will be 'family'. The keyword and response will have the individual index codes 'mother' and 'more' respectively. Note that keyword set index codes are of particular importance when using dynamic commands, because these codes enable you to 'refer back' and add new

keywords or responses to already-existing keyword sets in whatever order may be appropriate, something which would otherwise be quite impossible.

4.3.3 Interaction of Index-Coded Commands with the Existing Script

In general index codes simplify the interaction of new commands with the existing script – as illustrated above, a new command with the same index code as an existing command of the same type will simply replace it. However index codes also provide a means of achieving various other possible interactions – for details see the Command Syntax Reference Guide (§6.1).

4.4 Giving Direction to a Conversation

The system's facility for Phrase Memorisation and Recall (§2.4) can be made far more powerful and flexible through the use of Command Index Codes (§4.3) for memorised phrases.

4.4.1 Memorising Pronoun References

Suppose, as a simple example, that you are designing a system to discuss British politics, with responses that frequently make reference to the various political parties. In this case, the user is highly likely to input phrases containing the pronoun 'they', and it would clearly be helpful if the system were able to identify whom 'they' refers to in each case. This can be achieved by the following sort of method:

```
Klabour LABOUR
Klabour PEOPLE'S PARTY
Rlabour DO YOU UNDERSTAND LABOUR'S POSITION ON EUROPE?
        & {Mtheymem labour}
Rlabour WHAT DO YOU THINK OF THE LABOUR PARTY?
        & {Mtheymem labour}

Kconser CONSERVATIVES
Rconser WHAT DO YOU THINK OF THE CONSERVATIVES?
        & {Mtheymem the conservatives}
```

This creates two keyword sets, one with two keywords and two responses, and the other with one of each. But notice that each of the responses has an action associated – this has the effect of memorising an appropriate phrase (either 'labour' or 'the conservatives') under the index code 'theymem'. Hence at any point, the current value of that phrase will 'remember' which of the two parties was last mentioned in these responses. This memory can then be exploited by adding an input transformation to the Script:

```
I they => [Mtheymem]
```

Now if the user gives the input 'they are dreadful', this will immediately be converted into either 'labour are dreadful' or 'the conservatives are dreadful', depending on the current value of the phrase in question. Hence the Script is able to keep track of which party is being referred to in the user's input.

What happens here if nothing has yet been remembered under the index code 'theymem', because none of the relevant keyword responses has yet been 'triggered'? In this situation, as mentioned in §2.4, 'Phrase Memorisation and Recall', any input transformation which relies on that as-yet-non-existent memory will simply be ignored. Likewise, any void input or no-keyword response (etc.), or any output/final transformation or keyword response, which incorporates any not-yet-memorised phrase, will also be ignored. This feature of the system is destined to play a major role in the next section.

4.4.2 Changing Subject or Temper – First Method

Kenneth Colby's PARRY is a famous chatbot program designed to mimic the behaviour of a paranoid, who, as the conversation proceeds, usually gets progressively more agitated and annoyed (by reacting in this way to any input that can be interpreted as at all threatening, disbelieving, or hostile). In order to simulate this sort of change of temper, you can build in a variety of different responses, each of which makes reference to a memorised phrase whose index code is intended to represent the 'temperature' of the conversation at which it should be applied. For example:

```
Kbel BELIEVE
Rbel [Mcalm] DO YOU BELIEVE WHAT I'M SAYING?
Rbel [Magit] I DON'T THINK YOU BELIEVE ME!
Rbel [Manoy] I'M ANNOYED THAT YOU DON'T BELIEVE ME.
Rbel [Mpara] YOU'RE JUST PART OF THE CONSPIRACY, AREN'T YOU!
```

Here the crucial point is that the first response can be selected only if something has already been memorised under the index code 'calm', likewise the second can be selected only if a phrase has been memorised under the code 'agit', and so on for the others. But notice also that although the memorised phrase is actually part of the response in each case, *there is no requirement that the memorised phrase actually contain any characters*. It is therefore sufficient, for the first response to be made available, that the following command should have been executed at some point either in the original Script or dynamically:

```
Mcalm
```

This command memorises a 'phrase' under the index code 'calm', but the memorised phrase is the null string – nothing at all. Hence when it is incorporated into the first of the keyword responses, that response will appear as simply:

```
DO YOU BELIEVE WHAT I'M SAYING?
```

To put all this together, you could include in your Script the following commands:

```
Mcalm
Kbel BELIEVE
Rbel [Mcalm] DO YOU BELIEVE WHAT I'M SAYING?
     & {Mcalm\
        Magit}
Rbel [Magit] I DON'T THINK YOU BELIEVE ME!
     & {Magit\
        Manoy}
Rbel [Manoy] I'M ANNOYED THAT YOU DON'T BELIEVE ME.
     & {Manoy\
        Mpara}
Rbel [Mpara] YOU'RE JUST PART OF THE CONSPIRACY, AREN'T YOU!
```

The Script itself memorises the null string under the index code 'calm' only. Hence when the user inputs a sentence containing the keyword 'believe', the first response 'DO YOU BELIEVE WHAT I'M SAYING?' is the only one available for selection and display. When it is displayed, however, it simultaneously deletes the memorised phrase with the code 'calm', and instead memorises the null string under the index code 'agit' (intended to signify a level of agitation). So the next time the keyword 'believe' is matched, it is the second response 'I DON'T THINK YOU BELIEVE ME!' which is selected and displayed. When this is displayed, it deletes the 'agit' phrase and saves the null string under the code 'anoy', and so the process goes on.

In practice, of course, you are likely to want the 'temper' of the program to change in response to more than just a single keyword. But the techniques illustrated here are generalisable in obvious ways, enabling you to explore the system's potential for designing a conversational system capable of evincing a plausible temperamental progression, or making a reasonable pretence of following a sequence of topics. The latter is only really feasible if the topic sequence is fairly predictable – so here an appropriate challenge might be to attempt the sort of paranoid sequence pursued by Colby's PARRY program, where the mention of horses is used as a pretext for talking of horseracing and bookies, after which every opportunity is seized to lead the conversation on in turn through gangsters and rackets to the mafia (with money, gambling, the police and Italians also featuring as useful entry topics).

4.4.3 Changing Subject or Temper – Second Method

In §4.4.2, different responses were made available depending on *Elizabeth's* 'state', taking advantage of the fact that a response which contains a memory can be selected only if that memory exists. However *Elizabeth* also provides an alternative method of specifying 'conditional' behaviour, as exemplified by the following:

```
Mcalm
Kbel BELIEVE
<[Mcalm]>: Rbel DO YOU BELIEVE WHAT I'M SAYING?
```

```
            & {Mcalm\
               Magit}
<[Magit]>: Rbel I DON'T THINK YOU BELIEVE ME!
            & {Magit\
               Manoy}
<[Manoy]>: Rbel I'M ANNOYED THAT YOU DON'T BELIEVE ME.
            & {Manoy\
               Mpara}
<[Mpara]>: Rbel YOU'RE JUST PART OF THE CONSPIRACY, AREN'T YOU!
```

Here the condition has been put in angle brackets before the corresponding script command, but the behaviour will be exactly the same as in the previous section. The advantage of this method is to make the condition more explicit, making the Script somewhat easier to understand, especially when combined with the 'Show conditions in main tables' setting from the Analysis menu (as explained in §4.5, 'Defining and Using Conditional Commands').

4.4.4 Changing Subject or Temper – Third Method

A more sophisticated type of conditional behaviour is also possible, which tests not only for the *existence* of particular named memories, but also their *value*. This can be used to generate the same kind of behaviour as in the previous two sections, but more elegantly, as follows:

```
Mtemper CALM
Kbel BELIEVE
<[Mtemper]==CALM>: Rbel DO YOU BELIEVE WHAT I'M SAYING?
                   & {Mtemper AGIT}
<[Mtemper]==AGIT>: Rbel I DON'T THINK YOU BELIEVE ME!
                   & {Mtemper ANOY}
<[Mtemper]==ANOY>: Rbel I'M ANNOYED THAT YOU DON'T BELIEVE ME.
                   & {Mtemper PARA}
<[Mtemper]==PARA>: Rbel YOU'RE JUST PART OF THE CONSPIRACY, AREN'T YOU!
```

Here we have a single memory called *temper*, which acts as a 'state variable' by taking in turn the values 'CALM', 'AGIT', 'ANOY', and 'PARA'. The advantage of this method is that now the old memory is automatically overwritten whenever a new one is saved, so fewer commands are needed to achieve the same effect of a 'change of state'. For more advanced examples along similar lines, see §4.5, 'Defining and Using Conditional Commands', and §4.6, 'Implementing Propositional Logic'.

4.5 Defining and Using Conditional Commands

The section on 'Giving Direction to a Conversation' (§4.4) introduced the idea of conditional responses, whereby the behaviour of the system depends upon which memories are currently present. In this section (and also §4.6 on 'Implementing Propositional Logic'), we shall see how this idea can be taken much further. First, however, a quick summary of how conditions may be defined and viewed.

4.5.1 Defining Command Conditions

Virtually any script command may be preceded with a condition, which in the Script is always enclosed within angle brackets immediately followed by a colon, for example:

```
<[Mrugby]>: O target => TRY

<[Mfootball]>: O target => GOAL
```

The first of these defines an output transformation which will be invoked only if there is a 'rugby' memory, while the second defines a similar transformation which will be invoked only if there is a 'football' memory – these might be used in conjunction with a keyword transformation which gives a response such as 'I'D LIKE TO SCORE A target', to ensure that the appropriate 'target' is substituted.

An alternative way of producing this sort of behaviour is to use a specific memory which takes different values depending on the current topic, for example:

```
<[Mgame]==RUGBY>: O target => TRY
<[Mgame]!=RUGBY>: O target => GOAL
```

The first of these defines an output transformation which will be invoked only if there is a 'game' memory whose current value is 'RUGBY'; the second defines an output transformation which will be invoked only if there is a 'game' memory whose value is *something other than* 'RUGBY'.

Note also that conditions can involve more than one memory. Thus for example the transformation:

```
<[Mrugby] [Mfootball]>: O target => TRY OR GOAL
```

will operate only if *both* a 'rugby' *and* a 'football' memory are present, while the no-keyword response:

```
<[Mforename] [Msurname]==fred bloggs>: N PLEASE GIVE ME YOUR REAL NAME.
```

will be used only if both 'forename' and 'surname' memories exist, and if concatenated together they make the phrase 'fred bloggs'. If you make use of this sort of test, be sure to get the case right – user input will normally be saved in lower case unless you use the function UPPER (explained in §6.2, 'Built-In String Functions') to convert it.

If a condition makes reference to a memory that does not exist, then normally the condition will be treated as *false*, and hence the relevant script command will be ignored. However if the condition finishes with a question mark '?', then the opposite will apply – if any memory to which the condition refers does not exist (*or*, indeed, if the condition has been incorrectly defined, for example by missing out one of the square brackets), then the condition will be judged as *true*. An example might be a situation in which some default behaviour is required but is subject to modification if a conflicting memory exists:

```
<[Mlanguage]==english?>: N YOUR FIRST LANGUAGE IS ENGLISH, RIGHT?
```

This will treat the given phrase as a no-keyword response if *either* there is a 'language' memory with the value 'english', *or* there is no such memory at all. If the 'language' memory exists with a different value such as 'french', however, then the no-keyword command will be ignored.

Another use for this question mark facility is to prevent responses from being repeated without having to delete them, by assigning each such response a specific memory that 'blocks' it. Thus a response to a keyword might be defined as follows:

```
<[Mimnot]==ok?>: R I'M NOT GOING TO SAY THIS AGAIN!
                 & {Mimnot}
```

This response is not available unless *either* memory `[Mimnot]` does not exist *or* its value is 'ok'. When the response is given, however, it immediately creates memory `[Mimnot]` with a null value, ensuring that the response then becomes unavailable. It can later if necessary be revived *either* by deleting `[Mimnot]` entirely, or by setting it to 'ok', the latter perhaps being preferable because it retains the memory as a record that the response has been used. (Obviously the same sort of repetition blocking can be achieved by initialising the script with a set of memories which are then successively *deleted* to prevent responses' being used again. But the great advantage of the method shown here, which instead involves *adding* memories, is that no initial memory creation is required, enabling the keyword/response sets to be more self-contained and the script more 'modular'.)

4.5.2 Viewing Command Conditions

Command conditions may be viewed in either of two ways, depending on the setting of 'Show conditions in main tables' within the Analysis menu. If this item is unchecked (i.e. not ticked), then the only evidence of a command condition within the relevant table will be a '[Click]' entry in the final column of the table (which will be labelled 'Cond/Act' rather than 'Action' if any script commands have conditions attached). When you then click on the '[Click]' cell, the relevant condition will be displayed (together with any action applying to the same operation).

If on the other hand 'Show conditions in main tables' has been selected within the Analysis menu (i.e. is shown ticked), then any command condition will be shown within the main body of the relevant table, for example:

```
[Mrugby]  :  target
[Mforename] [Msurname]==FRED BLOGGS  :  PLEASE GIVE ME YOUR REAL NAME.
```

In this case *every* command in every table will be preceded by a colon (so the setting will be immediately apparent), but nothing will appear before the colon unless a corresponding condition has been defined.

4.5.3 Implementing a Questionnaire

Questionnaires are often suitable for straightforward conditional treatment, because their topic sequence is highly predictable, so they can be implemented quite effectively using extensions of the basic technique illustrated below. The following Script is in the file `Questionr.txt`, so you can easily test it out and see what it does. (Note that some of these text lines are so long that they are likely to reach the end of the Help window and 'word-wrap' to the next line – when you prepare your own Script in a text editor, make sure that such lines do indeed word-wrap if necessary, and are not split incorrectly by real line breaks.)

```
/ First a welcome message to get the questionnaire going

W PLEASE TELL ME YOUR NAME
  & {Mstate ASKNAME}
/ And a simple void input response and quitting message
V WELL?
Q BYE!
```

The memory `[Mstate]` acts as a 'state variable' which keeps track of what point has been reached in the questionnaire. When the initial welcome message appears, `[Mstate]` is set to the value 'ASKNAME', which indicates that the questionnaire is right at the beginning, asking for the user's name.

```
/ The next keyword set prevents the conversation from continuing after
the name and address have already been given and confirmed
<[Mstate]==FINISH>: K [X?]
 R I've got your details now - goodbye!
 R Please could you go now?
```

This first keyword will always be matched if `[Mstate]` has the value 'FINISH'. Hence once that value has been assigned (i.e. when the questionnaire is completed), the system will just give the two shown responses in turn until the user quits.

```
/ The next keyword set deals with confirming words
K YES
K CORRECT
K RIGHT
 <[Mstate]==CONFIRMNAME>: R Good. Now please tell me your Oxford
address.
                          & {Mstate ASKADDR}
 <[Mstate]==CONFIRMADDR>: R Excellent. That's it, now - goodbye!
                          & {Mstate FINISH}
```

The same set of confirming words is available for two distinct purposes – either to confirm the user's name, or the user's address. Depending on which has been confirmed, `[Mstate]` is moved forward either to 'ASKADDR' (i.e. the state of asking for an address) or to 'FINISH'. But unless the questionnaire is in a state of waiting for one of these types of confirmation (i.e. unless `[Mstate]` is either 'CONFIRMNAME' or 'CONFIRMADDR'), this keyword set will be ignored.

```
/ Now for a set of disconfirming words
K NO
K INCORRECT
```

```
K WRONG
 <[Mstate]==CONFIRMNAME>: R Let's start again, then. Please give me your
first name followed by your surname.
                          & {Mstate ASKNAME}
 <[Mstate]==CONFIRMADDR>: R Let's do that again, then. Please give me
your full Oxford address.
                          & {Mstate ASKADDR}
```

Likewise, the same set of disconfirming words is available for both name and address, and again, these will be acted on only if `[Mstate]` is either 'CONFIRMNAME' or 'CONFIRMADDR'. But disconfirmation, unlike confirmation, takes the state of the questionnaire back rather than forward, to 'ASKNAME' and 'ASKADDR' respectively.

```
/ The next keyword will respond only to input of exactly two words
K [] [word1] [word2] []
 <[Mstate]==ASKNAME>: R OK - so your first name is [word1] and your
surname is [word2]?
                      & {Mstate CONFIRMNAME
                         Mfirstname [word1]
                         Msurname [word2]}
```

This keyword set is ignored unless `[Mstate]` has a value of 'ASKNAME', and it uses a very crude test – only an input of exactly two words will be accepted as a name here.

```
/ The next keyword set identifies an Oxford address plus postcode.
/ It assumes that the address starts with a number.
K [number] [X] [!1?] OXFORD [!2?] OX[dig1][dig2?] [dig3][let1][let2]
K [number] [X] [!1?] OXFORD [!2?] OX[dig1][dig2?][dig3][let1][let2]
 <[Mstate]==ASKADDR>: R Fine. I have your name as [Mfirstname]
[Msurname] and your address as [number] [X] , OXFORD, OX[dig1][dig2?]
[dig3][let1][let2]
                      & {Mstate CONFIRMADDR
                         Maddress [number] [X] , OXFORD
                         Mpostcode OX[dig1][dig2?] [dig3][let1][let2]}
```

This keyword set is ignored unless `[Mstate]` has a value of 'ASKADDR'. It expects an Oxford address to begin with a number, and to contain the word 'OXFORD' followed (possibly after punctuation) by a postcode of the form 'OX1 2AB' or 'OX12 3AB' (with or without a space). If it succeeds in identifying an address plus postcode, it changes `[Mstate]` to 'CONFIRMADDR' to ask for the entire address to be confirmed. Note the use of `[!1?]` and `[!2?]` to match with any 'spare' punctuation before and after the word 'OXFORD': if instead `[!?]` was used for both occurrences, then the keyword would only match if the same punctuation was used in both places.

```
/ The next keyword is designed to identify an Oxford address with no
postcode
K [number] [X] [!?] OXFORD
 <[Mstate]==ASKADDR>: R OK - so your Oxford address is [number] [X], but
what's your postcode?
                      & {Mstate ASKPOST
                         Maddress [number] [X] , OXFORD}
```

If the previous keyword set failed to identify an Oxford address plus postcode, this one attempts to identify an Oxford address minus the postcode. If it succeeds, it changes `[Mstate]` to 'ASKPOST' to ask for the postcode separately.

```
/
```

```
/ The next keyword is designed to identify an Oxford postcode.
K OX[dig1][dig2?] [dig3][let1][let2]
K OX[dig1][dig2?][dig3][let1][let2]
 <[Mstate]==ASKPOST>: R Fine. I have your name as [Mfirstname]
[Msurname] and your address as [Maddress], OX[dig1][dig2?]
[dig3][let1][let2]
                      & {Mstate CONFIRMADDR
                         Mpostcode OX[dig1][dig2?] [dig3][let1][let2]}
```

This keyword set operates only if the previous set has identified an Oxford address minus postcode (so `[Mstate]` is 'ASKPOST'), and serves to identify a postcode by itself (of the form 'OX1 2AB' or 'OX12 3AB', with or without a space). If it succeeds in identifying a postcode, it changes `[Mstate]` to 'CONFIRMADDR' to ask for the entire address to be confirmed.

```
/ The next keyword points out when an 'OX' string doesn't appear to be a
valid postcode
K OX[term?]
 <[Mstate]==ASKPOST>: R That doesn't look like a valid Oxford postcode
to me. Please could you give it again?
```

If an 'OX' string has been entered in response to the request for a postcode, but the string isn't of the right form for a valid postcode, then this asks for it to be corrected.

```
/Finally, some no-keyword responses to deal with anything that hasn't
already been picked up
<[Mstate]==ASKNAME>: N Please just tell me your first name followed by
your surname.
<[Mstate]==CONFIRMNAME>: N Is your name [Mfirstname] [Msurname] - just
yes or no, please?
<[Mstate]==ASKADDR>: N Your full Oxford address, please.
<[Mstate]==ASKPOST>: N Please just tell me your Oxford postcode.
<[Mstate]==CONFIRMADDR>: N Is your address [Maddress], [Mpostcode] -
just yes or no, please?
```

This 'catchall' set of no-keyword responses provides an appropriate prompt, depending on the state of the questionnaire, whenever the user's input cannot be dealt with by one of the keyword sets.

Note that in this script the phrase memorisation is used for two distinct purposes – first, to keep track of the stage that the conversation has reached (while the `[Mstate]` memory moves through the sequence 'ASKNAME', 'CONFIRMNAME', 'ASKADDR', 'ASKPOST', 'CONFIRMADDR', and eventually 'FINISH'); secondly, to record the information given by the respondent (firstname, surname, address, and postcode).

4.5.4 A Final Warning

All this should prove to be interesting and enjoyable, but do not expect too much from it. Playing with the system can serve as a valuable introduction to some of the main concepts of Artificial Intelligence (AI) and Natural Language Processing (NLP), such as pattern matching, substitution, grammars, production rules, and expert systems. But in the end, it simply is not possible to produce a genuine language-understanding system without building in some genuine *understanding* – do not, therefore, be deluded into supposing that a conversational Script is ultimately any more than an amusing and instructive pretence (and don't spend hundreds of hours trying to perfect one). You can learn a tremendous amount by playing with this sort of system, but a time will come when to make more progress, you simply have to move on to the more serious business of AI and NLP theory, where the box of tricks becomes far more extensive, and far more informed by real-world understanding.

4.6 Implementing Propositional Logic

Although the system has not been designed with such sophisticated processing particularly in mind, the implementation of resolution theorem-proving within propositional logic provides an interesting and illuminating challenge and test case. The techniques described below are accordingly presented primarily as an illustration, to show how such things as sequential processing of multiple inputs, and logical interaction between them, can be implemented within *Elizabeth*. The following Script is in the file `Resolution.txt`. (Note that some of these text lines are so long that they are likely to reach the end of the Help window and 'word-wrap' to the next line – when you prepare your own Script in a text editor, make sure that such lines do indeed word-wrap if necessary, and are not split incorrectly by real line breaks.)

```
/C Recursion ON
/C Matchlimit 5000
/C Memchecklimit 500
/C Input Iteration unlimited
/C Input Cycling unlimited
```

Recursion will be necessary, and also repeated application of input transformations.

The example 'Problem of Evil' argument

```
W INPUT 'EG', OR AN ARGUMENT USING PROPOSITIONAL LETTERS, 'NOT', 'AND',
'OR', 'IMPLIES', 'THEREFORE', AND PUNCTUATION BETWEEN PROPOSITIONS.

I eg. => G implies (O and B) ; B implies T ; (O and T) implies not(E) ;
E ; therefore not(G).
```

The Script is designed to respond to an argument expressed using propositional letters and the standard logical connectives, with 'therefore' being used to indicate the conclusion. As an illustration, the input 'eg' will be replaced by a symbolic version of the classical Problem of Evil argument, which makes use of all of the Script's processing functions. This gives a famous challenge to religious belief: If God exists (G), then He is omnipotent (O) and benevolent (B); if God is benevolent (B), then He will try to eliminate evil (T); if God is omnipotent (O) and He tries to eliminate evil (T), then evil will not exist (not(E)); but evil does exist (E); therefore God does not exist (not(G)). The argument is valid, proving that anyone who believes in God needs to find fault with at least one of its premisses (e.g. by denying that a benevolent God will necessarily try to eliminate evil).

Final testing for validity and invalidity

```
<[M-1]==$>: F [x?] => THE ARGUMENT IS VALID
  & {!M-1 [x?]
     !F\}
```

As we'll see later, the argument is proved to be valid if at some point a single '$' is generated as a memory. The final transformation here tests whether the last numbered memory to be saved was '$', and if so, it replaces the active text (whatever that is, since `[x?]` will match anything) with the message 'THE ARGUMENT IS VALID', and saves the active text in place of the '$' memory before deleting all final transformations (including itself), which has the effect of terminating all processing. Note that `[M-1]` refers to the last-but-one memory in index order – this is needed because the last one will always be `[Mstage]` (since 's' comes later than the digits in the ASCII ordering).

```
H THE ARGUMENT APPEARS TO BE INVALID
```

If the argument is never shown to be valid, then eventually the system will terminate when it exceeds the match algorithm limit. This command defines an appropriate halting message, indicating that the argument appears to be invalid.

STAGE 1: replacing brackets, and separating propositions

The `[Mstage]` memory is used throughout to keep track of the stages of processing. Here it starts off being made equal to '1':

```
Mstage 1
```

The following three input transformations all have as a condition that `[Mstage]` should be equal to '1'; hence they are all confined to the first stage of processing and will be ignored as soon as `[Mstage]` moves on to higher values:

```
<[Mstage]==1>: I ( => <
<[Mstage]==1>: I ) => >

<[Mstage]==1>: I [X] [!] [x?] => [x?]
               & {!M [X]}
```

The first two input transformations here simply replace any curved brackets in the input with angle brackets – for most of what follows, curved brackets will be reserved for the purpose of indicating structured formulae that have been fully analysed. The third input transformation separates out the first proposition in the input, up to the first punctuation mark (because `[!]` matches any punctuation, and `[X]` is minimally matched): this proposition is saved, and the rest of the input is iterated through the same transformation until it has all been dealt with (i.e. when `[x?]` matches nothing after the final punctuation).

By the end of Stage 1 in processing our example Problem of Evil argument, the active text will be null and the system will contain the following memories:

001	G implies < O and B >
002	B implies T
003	< O and T > implies not < E >
004	E
005	therefore not < G >
stage	1

Stage 1 ends when the active text becomes null, so that the three input transformations above no longer have anything to do. Processing then moves on to a fourth transformation, which replaces the null text with '%END%':

```
<[Mstage]==1>: I [] => %END%
```

Moving on from each processing stage to the next

.From now on, the saved propositions and '%END%' will be 'rotated', each in turn becoming the active text – and thus made available for processing – before being re-saved when that processing is completed. Each subsequent processing stage finishes when all the propositions have been rotated, so that '%END%' again appears as the active text. The actual change of stages is achieved using two keyword transformations:

```
K %END%
 <[Mstage]==5a>: R %END%
                 & {!Mstage 5}
 <[Mstage]==8>:  R %END%
                 & {!Mstage 7}

K %END%
R %END%
  & {!Mstage [inc:[Mstage]]}
```

The first keyword transformation moves from Stage 5a to 5, and from 8 to 7 (we'll see why later). In all other cases, the second transformation simply increments `[Mstage]`, from 1 to 2, 2 to 3, 3 to 4, and so on. So when this is applied at the end of Stage 1, `[Mstage]` will be made equal to '2', thus moving us on to Stage 2.

Rotating propositions in Stages 2 to 8

After each proposition (as the active text) has been fully dealt with, processing moves on to a final transformation which achieves the rotation of propositions described above:

```
F [x] => {[M+1]}
  & {!M+1\
     !M [x]}
```

This transformation replaces the current proposition `[x]` with the first existing memory – `[M+1]` – which will be the next proposition to be rotated – this is recursed so that processing can start again from the relevant input transformations. Meanwhile that memory is deleted, and the replaced proposition `[x]` is itself saved (thus becoming the last numbered memory).

Removing superfluous brackets

The next few processing stages can at various points generate superfluous brackets, as the propositional structures are built up. These two input transformations are designed to remove any such unecessary brackets:

```
I (([b1])) => ([b1])
I <([b1])> => ([b1])
```

STAGE 2: identifying atomic propositions and logical operators, and negating the conclusion

All of the following input transformations operate only at Stage 2 – that is, when `[Mstage]` has a value of '2':

```
<[Mstage]==2>: I [word] => (^[word])
```

This places a caret (or circumflex) character '^' in front of each 'atomic proposition' in the argument ('G', 'O' etc.) and curved brackets around them.

```
<[Mstage]==2>: I (^not) => not
<[Mstage]==2>: I (^and) => and
<[Mstage]==2>: I (^or) => or
<[Mstage]==2>: I (^implies) => implies
<[Mstage]==2>: I (^therefore) => therefore
```

These remove the caret and brackets from the logical operators, so they are not confused with propositions. From now on, curved brackets will indicate a proposition whose structure has been fully analysed.

```
<[Mstage]==2>: I therefore [x] => not<[x]>
```

The method of 'resolution refutation' operates by attempting to prove a contradiction between the premisses of an argument and the <u>negation</u> of its conclusion – if there is such a contradiction, then the truth of the premisses must guarantee the falsehood of that negation (and hence, the truth of the unnegated conclusion). The above input transformation accordingly negates the argument's conclusion, by replacing 'therefore' with 'not<>'.

By the end of Stage 2 in processing our example Problem of Evil argument, the active text will be '%END%' and the system will contain the following memories:

007 (^G) implies < (^O) and (^B) >
008 (^B) implies (^T)
009 < (^O) and (^T) > implies not (^E)

010 (^E)
011 not < not (^G) >
stage 2

STAGE 3: analysing the logical structure of compound propositions

Stage 3 involves four input transformations which built up the structure of compound formulae 'from the inside out', by identifying logical operators that apply to already-analysed propositions (i.e. those that are already within curved brackets):

```
<[Mstage]==3>: I not ([b1]) => (NOT([b1]))
<[Mstage]==3>: I ([b1]) and ([b2]) => (([b1]) AND ([b2]))
<[Mstage]==3>: I ([b1]) or ([b2]) => (([b1]) OR ([b2]))
```

Notice how in these transformations, the relevant lower-case operator is changed to upper-case as the new outer brackets are added – this ensures that the transformation won't be re-applied to the same operator. Also note the use of the `[b…]` pattern, which guarantees that brackets are correctly matched within the relevant text.

```
<[Mstage]==3>: I ([b1]) implies ([b2]) => ((NOT([b1])) OR ([b2]))
```

This fourth transformation reflects the fact that in resolution theorem proving (as often in other formal logic contexts), 'P implies Q' is to be treated as 'not(P) or Q'.

By the end of Stage 3 in processing our example Problem of Evil argument, the active text will be '%END%' and the system will contain the following memories:

013 ((NOT (^G)) OR ((^O) AND (^B)))
014 ((NOT (^B)) OR (^T))
015 ((NOT ((^O) AND (^T))) OR (NOT (^E)))
016 (^E)
017 (NOT (NOT (^G)))
stage 3

STAGE 4: simplifying the logical structure of compound propositions

Resolution requires that all compound formulae be 'translated' into 'clausal form' (in which 'NOT' and 'OR' are the only logical operators) – most of this translation takes place through the following input transformations:

```
<[Mstage]==4>: I NOT(NOT([b1])) => [b1]
```

This removes 'double negations'.

```
<[Mstage]==4>: I NOT(([b1]) OR ([b2])) => (NOT([b1])) AND (NOT([b2]))
<[Mstage]==4>: I NOT(([b1]) AND ([b2])) => (NOT([b1])) OR (NOT([b2]))
```

These 'push NOTs inwards'.

```
<[Mstage]==4>: I ([b1]) OR (([b2]) AND ([b3])) => (([b1]) OR ([b2])) AND
(([b1]) OR ([b3]))
<[Mstage]==4>: I (([b2]) AND ([b3])) OR ([b1]) => (([b2]) OR ([b1])) AND
(([b3]) OR ([b1]))
```

These implement the rule of 'distribution' of OR over AND.

By the end of Stage 4 in processing our example Problem of Evil argument, the active text will be '%END%' and the system will contain the following memories:

019 (((NOT (^G)) OR (^O)) AND ((NOT (^G)) OR (^B)))
020 ((NOT (^B)) OR (^T))
021 (((NOT (^O)) OR (NOT (^T))) OR (NOT (^E)))
022 (^E)
023 (^G)
stage 4

STAGES 5 and 5a: separating conjunctive formulae into clauses

After Stage 4, the only 'AND's remaining should lie between independent clauses, which now need to be separated out. The following input transformation divides any 'AND' formulae accordingly:

```
<[Mstage]==5>: I [] ([b1] AND [b2]) [] => [b2]
               & {!M [b1]
                  !Mstage 5a}
```

This transformation uses the pattern `[]` to signify both the beginning and the end of the active text, thus ensuring that it will only match with an entire conjunctive proposition (i.e. a formula whose main operator is 'AND'). Any such proposition is then divided up, with the first 'conjunct' `[b1]` being put directly into memory while the second `[b2]` remains as the active text. There is, however, a complication, because if rotation continues as usual but the first conjunct is itself a conjunctive proposition, then this won't have been processed by the time '%END%' reappears and Stage 5 ends. For this reason, `[Mstage]` is set temporarily to '5a', so that the entire rotation of memories will be repeated a second time after Stage 5a has been changed back to Stage 5.

By the end of Stage 5 in processing our example Problem of Evil argument, the active text will be '%END%' and the system will contain the following memories – note that the propositions are now all in clausal form, with 'NOT' and 'OR' as the only remaining logical operators:

032 ((NOT (^G)) OR (^O))
033 ((NOT (^G)) OR (^B)))
034 ((NOT (^B)) OR (^T))
035 (((NOT (^O)) OR (NOT (^T))) OR (NOT (^E)))
036 (^E)
037 (^G)
stage 5

STAGE 6: simplifying the clausal representation

Stage 6 simplifies the clausal form representation, by removing the 'OR's, 'NOT's, and caret characters, and retaining only the brackets that immediately surround atomic propositions, changing these to angle brackets to indicate propositions that are negated.

```
<[Mstage]==6>: I (NOT(^[word])) => <[word]>
```

This replaces 'NOT's with angle brackets.

```
<[Mstage]==6>: I OR ([b1] OR [b2]) => OR [b1] OR [b2]
<[Mstage]==6>: I ([b1] OR [b2]) OR => [b1] OR [b2] OR
```

These remove internal bracketing within clauses.

```
<[Mstage]==6>: I [] ([b1] OR [b2]) [] => [b1] OR [b2]
```

This removes external bracketing around compound clauses.

```
<[Mstage]==6>: I OR =>
```

This eliminates 'OR's.

```
<[Mstage]==6>: I ^[word] => [word]
```

This eliminates any remaining carets.

By the end of Stage 6 in processing our example Problem of Evil argument, the active text will be '%END%' and the system will contain the following memories, all propositions now being in simplified clausal form:

039 < G > (O)
040 < G > (B)
041 < B > (T)
042 < O > < T > < E >
043 (E)
044 (G)
stage 6

STAGE 7: resolving the clauses together

So far we have been analysing the argument's propositions and, in some cases, dividing them up. But now our analysis is complete, and we face the more complicated task of comparing the analysed propositions together and drawing inferences from relevant pairs of them.

The resolution rule is logically very straightforward. Wherever one clause contains a 'positive literal' (i.e. an unnegated atomic proposition) and another contains the corresponding 'negative literal' (i.e. the same atomic proposition, but negated), we can infer from them a third clause which combines together all the elements of the two clauses except for these two literals. Thus for example from the two clauses (shown here in both conventional and simplified form):

not(P) or not(Q) or R *< P > < Q > (R)*

S or Q or not(T) *(S) (Q) < T >*

we may infer:

not(P) or R or S or not(T) *< P > (R) (S) < T >*

Recall that our aim in resolution refutation is to prove a <u>contradiction</u> from the combination of the argument's premisses with the <u>negation</u> of its conclusion, so in the case of our example argument, we want to show that the clauses represented by memories 039 to 044 listed above generate such a contradiction, which is manifested by the resolution rule when clauses such as:

T *(T)*

not(T) *< T >*

are used to infer the null clause (usually called 'NIL').

Before considering how this is implemented within Elizabeth, it might be helpful to see it applied to our actual example argument. We start with the clauses left in memory at the end of Stage 6:

1. < G > (O)
2. < G > (B)
3. < B > (T)
4. < O > < T > < E >
5. (E)
6. (G)

The simplest way of proceeding is to use a form of <u>unit resolution</u>, where we work through the 'unit clauses' (those that contain only a single 'literal'), in turn drawing from them whatever conclusions we can. In the current case, the first unit clause is 5, which can be resolved with clause 4 to generate:

7. < O > < T >

Then we move on to the only other unit clause, 6, which can be resolved with each of 1 and 2 to yield:

8. (O)
9. (B)

This gives us two more unit clauses to work with. The first of them, 8, resolves with each of 4 and 7 to give:

10. < T > < E>
11. < T >

while 9 resolves with 3 to give:

12. (T)

And now we notice that 11 and 12, which are both unit clauses, resolve together to produce:

13. NIL

thus demonstrating that the original six clauses were indeed inconsistent, and hence that our example argument is valid (because its premisses are inconsistent with the negation of its conclusion, so if all its premisses are true, then its conclusion must be true also).

This basic form of unit resolution can be implemented in Elizabeth using the following two rules, the first of which deals with positive unit clauses, and the second with negative unit clauses:

```
<[Mstage]==7>: I [] ([word]) [] => $ ([word])
  & {!<['Mstage]==7>: O [] [b1?] <[word]> [b2?] [] => [b1?] <[word]%
[b2?]
                       & {!M [b1?] [b2?]
                          !Mstage 8}}

<[Mstage]==7>: I [] <[word]> [] => $ <[word]>
  & {!<['Mstage]==7>: O [] [b1?] ([word]) [b2?] [] => [b1?] ([word]%
[b2?]
                       & {!M [b1?] [b2?]
                          !Mstage 8}}
```

When the first of these rules is applied to the active text:

(E)

the input transformation replaces this with:

$ (E)

but more significantly, it generates a new output transformation as follows:

```
!<[Mstage]==7>: O [] [b1?] <E> [b2?] [] => [b1?] <E% [b2?]
                & {!M [b1?] [b2?]
                   !Mstage 8}
```

It is this new output transformation that performs the resolution, by seizing any opportunity to interact with a clause containing '< E >'. So when clause 4 next appears as the active text:

< O > < T > < E >

this is converted to

< O > < T > < E %

which inhibits further resolution using this clause for the moment, while the result of the resolution:

< O > < T >

is stored in memory (this is clause 7 in our calculation above).

The second input transformation acts in a similar way, except that it applies when the active text is a negative literal, and creates an output transformation which itself lies in wait for clauses containing the corresponding positive literal.

STAGE 8: keeping the resolution process under control

The main practical problem when implementing resolution theorem proving is in keeping the resolution under control – in any complex argument, there are likely to be large numbers of clauses that can interact with each other, and it's vital to ensure that they don't go on resolving and resolving over and over again without moving forward towards the desired contradiction. The method used here involves two main control mechanisms. First, any unit clause is tagged with '$' once it has been used to generate the relevant output transformation – this prevents its having the same effect again (e.g. through input transformation iteration on the same text). Secondly, whenever any such output transformation is itself applied, the second bracket around the relevant literal is replaced with '%' to inhibit its being used again, and processing passes into Stage 8, ensuring that the rotation of propositions will be started again from the beginning after the system returns back to Stage 7. Apart from serving this purpose of rotation control, Stage 8 contains only two simple rules designed to restore corrupted brackets whenever the active text contains them – in practice, these rules will seldom be applied unless Stage 7 has been completely exhausted using the propositions available.

```
<[Mstage]==8>: I <[word]% => <[word]>
<[Mstage]==8>: I ([word]% => ([word])
```

To appreciate the importance of these control mechanisms, try changing the Stage 7 rules so that '%' is no longer inserted – you'll find that the example argument churns away purposelessly for ages, generating large numbers of superfluous memories.

And finally …

As each unit clause is used to generate a corresponding output transformation, it is tagged with '$' to prevent its being used for the purpose again, and then continues to be rotated. Eventually, if the argument is provably valid by this means, such a tagged unit clause will encounter an output transformation generated by its contradictory literal, and when this is applied, only the tag '$' will remain (hence '$' represents 'NIL'). It is this '$' that provides the test for validity, as we saw earlier.

4.6.1 *Elizabeth* and Theorem Proving

The method used above is not the only way of implementing resolution theorem proving within *Elizabeth*, and it is probably not the best. Certainly it's not the most powerful, since unit resolution is 'incomplete' – that is, there exist some sets of clauses, such as:

P or Q
P or not(Q)
not(P) or Q
not(P) or not(Q)

which are inconsistent, but which cannot be proved to be inconsistent by unit resolution (in the case of this set, because there are no unit clauses at all to work with). It is an interesting challenge to come up with a method which does not have this weakness (e.g. using the more powerful technique of 'linear resolution') but which at the same time remains sufficiently straightforward and controllable to be usable within *Elizabeth*. However it should be stressed that the main point of implementing resolution here has not been to assist with practical theorem-proving – if you aim to do theorem-proving as a serious business, then you'd be better advised to use a system designed for the purpose (such as Prolog). Rather, the point has been to illustrate some of the possibilities available within *Elizabeth's* dynamic rules, and to provide some insight into how these sorts of automated processes can make complex and purposeful computations such as theorem-proving possible. In this lies the fascination of the science of Artificial Intelligence, to which *Elizabeth* aspires to provide only a modest introduction.

4.7 Implementing a Turing Machine

In the 'Advanced script processing' submenu of the Help menu, there is a script named 'Turing machine for GCD', which implements a simple Turing machine that can calculate the greatest common divisor (or highest common factor) of two numbers. For example if the initial input is as follows:

```
0 1 1 1 1 0 1 1 0
```

This is to be interpreted as the number 4 followed by the number 2, while:

```
0 1 1 1 1 1 1 0 1 1 1 1 0
```

is to be interpreted as the number 6 followed by the number 4. In either of these cases, the output will be:

```
0 1 1 0.
```

showing that the greatest common divisor is 2.

This implementation of a Turing machine is actually quite simple, and has instructive similarities with Turing's own way of doing it in his famous 1936 paper 'On Computable Numbers, with an Application to the *Entschedungsproblem*'. Notice that at any point in processing, 'L' and 'R' are used to bracket the position of the read/write head (the 'current position'), while the memory named 'state' keeps track of the machine's state. Nearly all of the keyword transitions are conditional, depending on the current value of 'state', and each of these follows the pattern given by one of the three 'Templates' which are 'commented out' early in the script, showing how rightwards, leftwards, and stationary Turing machine operations can be achieved. Here, for example, is the template for a rightward movement:

```
/ Template for right
<[Mstate]==S>: K [x1?] L C R [n] [x2?]
 R {[x1?] c L [n] R [x2?]}
   & {!Mstate s}
```

Here 'S' is the name of the machine state of the Turing machine before the transition takes place, and 'C' is the character at the current position on the machine tape which triggers this transition when in state S. We are assuming here that the desired effect of the transition is to (a) write character 'c' back to the tape at the current position, (b) transition into machine state 's', and (c) move the current position to the right.

The keyword is conditionally defined, so it will only potentially come into play if the 'state' memory is indeed 'S', and it will only achieve a match if the character 'C' appears on the tape between 'L' and 'R'. If such a match occurs, '[n]' will be matched to whatever character appears just to the right of 'R'.

There is only one response, which:

- replaces 'L C R [n]' with 'c L [n] R' and recurses the resulting string, and
- forces immediate change of the 'state' memory from 'S' to 's'.

The effect of the string replacement is most easily seen if we imagine this in the context of a longer section of the machine tape, with character 'C' originally at position 4, and [n] at position 5 (remembering that 'L' and 'R' do not represent tape characters, but are simply there to indicate the position of the read/write head):

0 1 2 3 L C R [n] 6 7 8 9

After replacement, we have:

0 1 2 3 c L [n] R 6 7 8 9

So position 4 now contains 'c', and the read/write head has moved from position 4 to position 5 (thus moving to the right, as was intended).

Working out the rest is left as an exercise for the enthusiastic reader, with the aid of the comments embedded in the script.

5. Technical Details

5.1 Capitalisation and Transformations

Note that to see the effects discussed below in operation, you will need to look at the system's 'Trace' panel which displays the intermediate processing steps by which input is transformed into output. Usually the system displays its output in upper case only, although this can be changed using the Processing menu.

At stage 1 of the transformation process, all of the user's input is converted into lower case (so no capital letters remain). Then within the various transformation processes, the pattern matching process operates as follows:

(a) A lower-case pattern can match part of the active text only if that text is itself lower case.

(b) An upper-case pattern can match either upper- or lower-case text.

The clearest use of these conventions is in output (and final) transformations, where without them serious problems would arise. Suppose for example that the following keyword transformation is specified in the Script:

```
K I THINK [phrase]
 R WHY DO YOU THINK [phrase]?
```

Thus if the user inputs:

```
I think computers are stupid.
```

Then the system will respond:

```
WHY DO YOU THINK COMPUTERS ARE STUPID?
```

But now suppose that the matched phrase contains a first- or second-person reference, for example:

```
I think you are a computer.
```

Here a straightforward application of the keyword transformation will generate:

```
WHY DO YOU THINK YOU ARE A COMPUTER?
```

which is obviously inappropriate. But as explained with the Illustrative Script and Conversation in §1.4, there is a method of dealing with this sort of problem by specifying output transformations within the Script. This is where capitalisation plays a crucial role. For suppose first that the relevant output transformations were specified as follows:

```
O YOU ARE => I AM
O I AM => YOU ARE
```

Since these are entirely in upper case, the left-hand text in each transformation can match either upper- or lower-case text, so the processing will take place as follows:

`I think you are a computer.` (user input)

`i think you are a computer.` (converted into lower case)

`WHY DO YOU THINK you are a computer?` (keyword transformation)

`WHY DO YOU THINK I AM a computer?` (output transformation: 'YOU ARE' replaced by 'I AM')

`WHY DO YOU THINK YOU ARE a computer?` (output transformation: 'I AM' replaced by 'YOU ARE')

`WHY DO YOU THINK YOU ARE A COMPUTER?` (capitalised and output)

Hence the second output transformation just reverses the effect of the first one, defeating the point of having such transformations. If, however, the transformations are specified instead with a mixture of upper and lower case:

```
O you are => I AM
O i am => YOU ARE
```

then the processing will take place as follows:

`I think you are a computer.` (user input)

`i think you are a computer.` (converted into lower case)

`WHY DO YOU THINK you are a computer?` (keyword transformation)

`WHY DO YOU THINK I AM a computer?` (output transformation: 'you are' replaced by 'I AM')

`WHY DO YOU THINK I AM A COMPUTER?` (capitalised and output)

The crucial point here is that the second output transformation does not apply, because the left-hand text of that transformation, 'i am', being in lower case, does not match against 'I AM'.

5.2 Iteration and Cycling of Input/Output/Final Transformations

Input, output, and final transformations sometimes need to be applied repeatedly. Suppose, for example, that you have a transformation to 'collapse' sequences of the word 'very':

```
I very very => very
```

This will change, for example, 'I am very very happy' to 'I am very happy'. But suppose it is applied to the active text:

```
He is very very very very happy
```

The first application of the transformation will match the first two 'very's and the last two, reducing each pair to a single 'very' and thus yielding the result:

```
He is very very happy
```

But notice that the transformation could now be applied again to its own result – in other words, it could be 'iterated'. Normally the system does not iterate input, output, or final transformations, but you can set it to do so using the Control menu dialog. If you do this in the current instance, the active text will be transformed again automatically, to:

```
He is very happy
```

If iteration is selected, then the system will normally apply each input transformation repeatedly, only stopping when the transformation no longer 'matches' the active text or no longer makes any difference to it.

5.2.1 Limiting the Iteration of Transformations

This kind of iteration of input, output, or final transformations can cause problems if you are unwise enough to include certain types of 'non-terminating' transformations in the Script, illustrated by the following:

```
I tick => tick tock
```

If such a transformation is applicable to the active text, then it can be applied repeatedly for ever, because the string that results from the replacement will still contain the 'tick' pattern that triggers the transformation. Thus if the active text starts off as 'The clock went tick', this will progressively be changed to 'The clock went tick tock', 'The clock went tick tock tock', 'The clock went tick tock tock tock', and so on. To allow this to continue indefinitely would obviously cause the program to 'hang', so as explained in §3.3.1, *Elizabeth* imposes a limit on the number of times that the matching algorithm which underlies the various transformations can be called, this limit being adjustable from the Control menu dialog. By default, the *match algorithm limit* is 5,000.

Another option from the Control menu dialog gives an alternative method of iteration control, though one that is available only if recursion is disabled. This method acts according to the following rule (though the parameter of 10 is adjustable from the Control menu dialog):

> *If any input, output, or final transformation is applicable to the active text 10 or more times in succession, then it is applied just once.*

Hence if this rule is operational, the transformation above will just change 'The clock went tick' into 'The clock went tick tock', and then will not be applied again.

To sum up, the Control menu dialog provides three possibilities for limiting the iteration of input transformations, and the same three possibilities for output and final transformations. In the default situation, no iteration occurs – any transformation will be applied at most once to each part of the active text, though this is quite consistent with a transformation such as:

```
I dad => father
```

making more than one change if 'dad' occurs more than once in the active text (see §5.3, 'Technical Note on Pattern Matching', for more on how this works). One alternative (as explained in the previous paragraph) is to permit iteration, though with an automatic limit to allow any input, output, or final transformation to be applied a maximum of, say, 10 times repeatedly. The final possibility is to allow 'unlimited' iteration, with non-terminating transformations being halted only by the *match algorithm limit*.

5.2.2 Limiting the Cycling of Transformation Sequences

Input, output, and final transformations are arranged in sequence, and usually it is assumed that this sequence of transformations is to be applied just once – the system is expected simply to work through the list of transformations one by one, applying each of them in turn (with iteration if appropriate). However sometimes – most notably in the case of Implementing Grammatical Rules (§4.1) – you may wish to apply the entire sequence of transformations repeatedly, a process which is called 'cycling'.

Just as in the case of iteration, it is possible to produce non-terminating cases of cycling, for example:

```
I tick => tock
I tock => tick tack
```

Here if the active text starts off as 'The clock went tick', then this will be changed by the pair of transformations to 'The clock went tick tack'. Neither transformation by itself will iterate in this case (each of them only applies once at a time), but if the pair of transformations is repeatedly cycled, the active text will progressively be transformed to 'The clock went tick tack tack', 'The clock went tick tack tack tack', and so on. So exactly as with iteration, when cycling is enabled it is necessary to impose a limiting rule to prevent the program from hanging. Again the *match algorithm limit* plays a role, preventing any new cycling after that limit (5,000 by default) has been exceeded. And again there is an alternative method of control, available only if recursion is disabled. This method acts according to the following rule (and again the parameter of 10 is adjustable from the Control menu dialog):

> *If cycling is enabled, and the entire sequence of input, output, or final transformations is applicable to the active text more than 10 times in succession, then it is applied just once.*

Hence if this rule is operational, the sequence above will just change 'The clock went tick' into 'The clock went tick tack', and then will not be cycled.

So just as in the case of iteration, the Control menu dialog provides three possibilities for limiting the cycling of input, output, and final transformation sequences. In the default situation, no cycling occurs – the input transformation sequence is applied just once, and likewise the output and final transformation sequences. One alternative is to allow either sequence to be cycled, but with cycling permitted a maximum of, say, 10 times. The final possibility is to allow 'unlimited' cycling, with non-terminating cases being halted only by the *match algorithm limit*.

5.2.3 Setting Iteration/Cycling/Recursion Behaviour from a Script

The various iteration, cycling and recursion options that can be set interactively from the Control menu dialog can also be set within a Script, by making use of the '`/C`' directive. By far the easiest way of achieving this is first to set the desired behaviour interactively, and then before exiting from the Control menu dialog, click on the checkbox just underneath the 'OK' button, labelled 'Insert settings within script file'. After you've clicked on 'OK', your script file will start something like this:

```
/C Recursion ON
/C Matchlimit 5000
/C Memchecklimit 500
/C Input Iteration unlimited
/C Output Iteration unlimited
```

```
/C Final Iteration prevented
/C Input Cycling prevented
/C Output Cycling prevented
/C Final Cycling prevented
```

All of these can if you wish be hand-edited, but it's certainly safest to let *Elizabeth* make any changes for you to ensure that you've got the syntax right. Any Script that starts in this way will have the appropriate settings made automatically whenever it is run.

5.3 Technical Note on Pattern Matching

When you specify a pattern for any input, output or keyword transformation, the system automatically adds the term `[X#1?]` to the beginning of that pattern, and `[X#2?]` to the end, unless the pattern already begins or ends, respectively, with either a `[]` or an `[X]/[x]` term – in this case, assuming that 'Check final punctuation' is selected in the Processing menu, just `[.#?]` will be added to the end of the pattern if necessary to enable the final punctuation to be matched. This results in modifications such as the following:

`[phrase] []`	becomes	`[X#1?] [phrase] [.#?]`
`[] [phrase]`	becomes	`[phrase] [X#2?]`
`[phrase] [X]`	becomes	`[X#1?] [phrase] [X]`
`[X] [phrase]`	becomes	`[X] [phrase] [X#2?]`

These added terms are usually hidden in the relevant display tables (though they can be shown using 'Show hidden terms' from the Analysis menu), but they play a vital role in the system, ensuring that all matching can be done with the *entire* active text, because these two hidden terms can match the ends of that text, whatever those ends may be. (The reason why the particular patterns `[X#1?]` and `[X#2?]` are used for this purpose is that the hash symbol '#' is 'filtered out' of any user input or Script commands – as explained in §6.3, 'Alphabets and Special Symbols' – so there is no risk of any confusion with Script substitution patterns.)

As an example, suppose that you specify the input transformation pattern illustrated in §2.3 ('The Input/Keyword/Output/Final Transformation Process'):

```
I dad => father
```

Within the system, this is treated as the input transformation:

```
I [X#1?] dad [X#2?] => [X#1?] father [X#2?]
```

so given the input 'my dad is nice.', for example, `[X#1?]` will be matched with 'my' and `[X#2?]` with 'is nice.' – similar substitution in the right-hand side of the transformation then delivers the result 'my father is nice.'.

5.3.1 Multiple Matches of Input, Output, and Final Transformations

Compare the transformation just given with the user-defined:

```
I [X1?] dad [X2?] => [X1?] father [X2?]
```

It might seem that the operation of the two should be completely identical, but in fact they will differ if applied to the active text:

```
my dad is taller than your dad .
```

where the former will yield:

```
my father is taller than your father .
```

while the latter is applied once only to give:

```
my father is taller than your dad .
```

The system treats transformations to which the terms [X#1?] and [X#2?] have been added as potentially multiply-applicable to a single active text string. Other transformations are treated as only singly-applicable, since in these cases multiple applications would involve *explicitly* user-defined terms that overlap, so that such multiple applications would more appropriately be classed as iterations (i.e. repeated applications of the transformation to a progressively transformed active text) rather than simultaneous multiple matchings of a single active text. For details of how to control iterated application of input, output, and final transformations, see §5.2, 'Iteration and Cycling of Input/Output/Final Transformations'.

5.3.2 Matching and Search Order

Turning now to keyword transformations, suppose that you define a keyword and corresponding response as follows:

```
K [phrase1] IS BETTER THAN [phrase2]
 R DO YOU MEAN YOU PREFER [phrase1] TO [phrase2]?
```

With the automatically added terms described above, the keyword becomes

```
[X#1?] [phrase1] IS BETTER THAN [phrase2] [X#2?]
```

(the terms are added only to the keywords; responses always remain unchanged). If now the user inputs:

```
classical music is better than punk rock, don't you agree?
```

then [X#1?] will match with nothing (the null string), [phrase1] with 'classical music', [phrase2] with 'punk rock', and [X#2?] with ', don't you agree?'. So the corresponding response will be:

```
DO YOU MEAN YOU PREFER CLASSICAL MUSIC TO PUNK ROCK?
```

(Notice here how useful it is that the term [X#1?] can match with nothing, and that the term [phrase2] is prevented from matching with punctuation – without the latter restriction, the response would be 'DO YOU MEAN YOU PREFER CLASSICAL MUSIC TO PUNK ROCK, DON'T YOU AGREE?').

All this works correctly, but it is worth examining in detail *how* it works. To see why there's an issue here, notice that the sequence of terms

```
[X#1?] [phrase1]
```

can match against

```
classical music
```

in two different ways, the second possibility being when [X#1?] matches with 'classical' and [phrase1] with 'music'. Likewise, the sequence of terms

```
[phrase2] [X#2?]
```

can match against

```
punk rock, don't you agree?
```

in two different ways, the second possibility here being when [phrase2] matches with 'punk' and [X#2?] with 'rock, don't you agree?'. So given these alternatives, how does the system 'get it right' in both cases, and correctly match [phrase1] with 'classical music' and [phrase2] with 'punk rock'? The answer is simply this: *When seeking a match for an [X]-type term (with a capital 'X'), the system searches in increasing order, and therefore starts by trying the minimum possible match; but when seeking a match for a [phrase]-type term (lower-case 'p'), it instead searches in decreasing order, and hence starts by trying the maximum possible match.* (For a full explanation of the significance of upper- and lower-case terms, see 'Increasing or Decreasing Search' in §3.1, 'Pattern Matching'.) So whenever you have the combination of an [X] and a [phrase] field, in either order, the [phrase] field will always match as much as possible, which is why in the example above [phrase1] ends up matching 'classical music' rather than just 'music', and [phrase2] ends up matching 'punk rock' rather than just 'punk'.

5.4 Sequencing and Timing of Dynamic Commands

By default, dynamic commands are performed only *after* the user's input has been fully processed to generate an appropriate output. The order in which they are performed is as follows (and note that the actions triggered by substitution of a memorised phrase always precede the actions triggered by the string into which that substitution takes place):

1. Any actions triggered by input transformations, following the sequencing order of those transformations, i.e. by lexicographical (alphabetical) ordering of index codes;

(m) Any actions triggered by use of memorised phrases within the input transformations will take place immediately before any actions triggered by the corresponding input transformation.

2. Any actions triggered by the successful application of a keyword transformation. These are performed in the order:

(m) Actions triggered by the use of memorised phrases within the keyword transformation;

(k) Actions triggered by the (actually used) keyword;

(r) Actions triggered by the chosen response.

3. Any actions triggered by output transformations, following the sequencing order of those transformations, i.e. by lexicographical (alphabetical) ordering of index codes;

(m) Any actions triggered by use of memorised phrases within the output transformations will take place immediately before any actions triggered by the corresponding output transformation.

4. Any actions triggered by final transformations, following the sequencing order of those transformations, i.e. by lexicographical (alphabetical) ordering of index codes;

(m) Any actions triggered by use of memorised phrases within the final transformations will take place immediately before any actions triggered by the corresponding final transformation.

5. Any actions triggered by the choice of a specific void input or no-keyword response. These are performed in the order:

(m) Actions triggered by the use of memorised phrases within the void input or no-keyword response;

(r) Actions triggered by the response.

6. Deletion of any self-deleting messages, transformations, keywords or responses that have been used in the above processing (see §4.2.4, 'Self-Deletion').

5.4.1 Forcing Immediate Action

Any dynamic action can be made to occur immediately (i.e. without waiting for the output to be generated first), by prefixing the command letter with an exclamation mark '!'. Thus, for example, the action:

```
{!O you => them}
```

if performed by a keyword transformation, will immediately create an output transformation, potentially in time to affect the processing of the current response. Commands performed in this way are mutually sequenced in the same order as above. If in doubt, the actual sequencing of dynamic commands can be inspected very easily by consulting the Trace table. (Note that the possibility of immediate forcing is not applicable to automated self-deletion, which always takes place at stage 6 as explained above. However immediate self-deletion, if required, can be implemented using a dynamic deletion command rather than relying on the automatic facility.)

6. Reference Section

6.1 Command Syntax Reference Guide

For a general explanation of Script commands, see §2.1, 'Introduction to Script Commands', and the sections referenced from it. Note in particular that these commands can be called not only from a Script, but also dynamically, as explained in §4.2, 'Dynamic Script Processing'. Also note that the vast majority of Script commands can be associated with a further action (as explained in the same section) and can themselves be made conditional (as described in §4.5, 'Defining and Using Conditional Commands'). In what follows, these two complications will generally be ignored except for a brief final part of each section which summarises the relevant behaviour. Also ignored here is the possibility of *self-deletion*, whereby most of the commands below can be prefixed with '\' to make the relevant message, transformation, keyword or response usable once only (see §4.2.4, 'Self-Deletion', for details). In all other respects, the main sections below provide examples of virtually all the available types of command, designed to illustrate the various syntactical possibilities that are provided by the system.

6.1.2 Welcome, Quitting, Exit, Void Input and No-Keyword Messages

The function of these various types of message is explained in §2.2, 'Simple Message Selection Commands'. All of them have the same syntactic options, but they are illustrated below using no-keyword responses only (since these are in practice the most important of the five types):

`N GO ON`	add 'GO ON' to the set of no-keyword responses (and if a 'GO ON' response already exists, replace it)

Replacement, as mentioned in the bracket here, might seem pointless, because it leaves the text of the response completely unchanged. But it can nevertheless have a point if the new 'GO ON' response differs from the old one in some other way – for example, in being associated with a different action (actions are explained in §4.2, 'Dynamic Script Processing').

`N\ GO ON`	delete 'GO ON' from the set of no-keyword responses (if it is in that set); like most deletion commands, this deletes at most one instance, which will be the first match found as it searches through the responses (in index order)
`N\`	delete all no-keyword responses

Note that the command '`N`' by itself will generate an error message, because it is not possible to have a null no-keyword response (nor a null welcome message etc.).

`Ngo GO ON`	add 'GO ON' to the set of no-keyword responses, giving it the index code 'go' (and replacing any existing no-keyword response that already has 'go' as an index code)
`Ngo\`	delete the no-keyword response having the index code 'go' (if there is one)
`Ngo\ GO ON`	delete the no-keyword response having the index code 'go' (if there is such a no-keyword response), but only if its text is 'GO ON'

The difference between the last two of these is simply that the last makes two checks rather than one before deletion – it makes a deletion only if the set of no-keyword responses contains a response which *both* has the index code 'go' *and also* has the text 'GO ON'.

Conditions and Actions

No-keyword responses can be defined as conditional in the standard way, for example:

`<[Mbye]>: N BYE`	define a no-keyword response, 'BYE', which however will be usable only if the memory `[Mbye]` has been defined.

If you define such a conditional no-keyword response and later wish to delete or replace it, then you can repeat the condition in your deletion or replacement command to ensure that the right response is identified (though there is no need to include the condition unless there is a risk of ambiguity, and an index code usually provides a more convenient method of reidentification). So for example:

```
<[Mmem]>: N\ GO ON
```

will make a deletion only if a no-keyword response exists with the same text ‘GO ON’ and the same condition.

Actions associated with no-keyword responses have the obvious interpretation: they are performed whenever the response in question is used. So for example the response:

```
Npgo PLEASE GO ON
     & {Npgo\}
```

will delete itself as soon as it is used, thus ensuring that it is used only once. Note however that a similar effect could be achieved more simply by defining the response as self-deleting:

```
\N PLEASE GO ON
```

as in the example given in §2.2, ‘Simple Message Selection Commands’.

6.1.3 The Halting Message

The system has the facility of a single ‘halting’ message, which will be triggered if processing is halted through exceeding of the *match algorithm limit* (as explained in §3.3.1) or the *memory checking limit* (as explained in §6.1.4), for example:

`H I GIVE UP!`	define ‘I GIVE UP!’ as the system’s halting message

The halting message can also be defined interactively through the Control menu dialog, which provides a means for inspecting its current setting.

Conditions and Actions

Since the condition under which the halting message appears is fixed, any condition specified in association with it will be ignored. However an action can be included, for example:

```
H I GIVE UP!
  & {H I GIVE UP AGAIN!}
```

which will replace itself with a new halting message as soon as it has been used. The currently defined action can be inspected at any time through the ‘show halt action’ button in the Control menu dialog.

6.1.4 Memorised Phrases

The general function and behaviour of memorised phrases is explained in §2.4, ‘Phrase Memorisation and Recall’. The relevant syntactic options are very similar to those of the simple responses above, except that it is possible to memorise the null phrase, and also to delete memorised phrases by pattern matching:

`M OK`	add ‘OK’ to the set of memorised phrases (and if an ‘OK’ already exists, replace it)
`M`	add the null phrase (i.e. ‘’) to the set of memorised phrases (etc.)

As above, replacement can have the point of changing the action associated with the memorised phrase.

`M\ OK`	delete ‘OK’ from the set of memorised phrases (if it is in that set); like most deletion commands, this deletes at most one instance, which will be the first match found as it searches through the memorised phrases (in index order)
`M\ YES [X]`	delete any memorised phrases that match the pattern ‘`YES [X]`’ – in other words, that consist of ‘YES’ followed by some further text (for details of pattern matching, see §3.1); note that this kind of pattern deletion, which is only available with memorised phrases, is *not* restricted to deletion of only a single instance
`M\`	delete all memorised phrases
`M-0\`	delete the last memorised phrase (in index order)
`M-2\`	delete the last-but-2 memorised phrase (in index order)
`M+3\`	delete the third memorised phrase (in index order)

In practice, you are more likely to save and delete memorised phrases using index codes, to exploit the sorts of possibilities illustrated in §4.4, 'Giving Direction to a Conversation':

`Msaid OK`	memorise the phrase 'OK' under the index code 'said' (and if some other phrase has already been memorised under that index code, replace it)
`Msaid`	memorise the null phrase under the index code 'said' (etc.)
`Msaid\`	delete the memorised phrase having the index code 'said' (if there is one)
`Msaid\ OK`	delete the memorised phrase having the index code 'said' (if there is one), but only if the phrase itself is 'OK'
`Myes\ YES [X]`	delete the memorised phrase which has the index code 'yes' and matches the pattern '`YES [X]`' (if there is such a memorised phrase)

Again, the difference between the two '`Msaid\`' commands is simply that the second makes two checks rather than one before deletion – it makes a deletion only if the set of memorised phrases contains a phrase which *both* has the text 'OK' *and also* has the index code 'said'. (Although it is not possible to do this sort of double check in the case of the null phrase, this shouldn't be a problem, because you are unlikely to want to store null phrases except as contentless 'flags' to indicate the situation within a dialogue, as illustrated in §4.4, 'Giving Direction to a Conversation'.)

Conditions and Actions

If a memorised phrase is defined conditionally, then it will be recognised only if that condition is satisfied, hence for example:

```
<[Mone]>: Mtwo TWO
```

will enable `[Mtwo]` – and its stored value 'TWO' – to be used only if `[Mone]` is recognised. This could in turn be conditionally defined, in which case that condition also must be satisfied if `[Mtwo]` is to be used, and so on. Note that this could allow the possibility of a recursive loop of memory dependency, causing processing to hang. The system accordingly provides a *memory checking limit*, by default 500, which is set through the Control menu dialog like the related *match algorithm limit* (explained in §3.3.1).

If you define such a conditional memory and later wish to delete or replace it, then you can repeat the condition in your deletion or replacement command to ensure that the right memorised phrase is identified (though there is no need to include the condition unless there is a risk of ambiguity, and an index code usually provides a more convenient method of reidentification). So for example:

```
<[Mone]>: M\ TWO
```

will delete all, and only, those memorised phrases that have the text 'TWO' and the specified condition. As explained above, commands for the deletion of memorised phrases, unlike those for deletion of transformations, standardly allow for multiple (and also pattern-matching) deletion; index codes should be used if you wish to avoid this.

Actions associated with memorised phrases will be carried out whenever the phrase is used in processing, usually for substitution. For example:

```
Mtwo TWO
     & {Mtwoused}
```

However if two uses are made *simultaneously* (e.g. where '2 2' is changed to 'TWO TWO' through an input transformation that replaces '2' with `[Mtwo]` having the value 'TWO'), the associated action (i.e. the action associated with use of the memory `[Mtwo]`) will be performed once only. In cases of iteration or cycling, however (see §5.2, 'Iteration and Cycling of Input/Output/Final Transformations'), a memory can be used many times resulting in repeated actions, so care should be taken accordingly.

6.1.5 Saving Text to Files

Text can be saved dynamically to files, as explained in §2.5, 'Saving Text Files Dynamically'. The syntactic options here are similar to those for other commands (such as with memorised phrases), but there are significant differences in behaviour, for three main reasons. First, text saving commands don't usually change anything within *Elizabeth*'s internal structures (e.g. they don't create any new transformations or memories); secondly, they involve interaction with an external file; thirdly, for this reason they carry a risk of hardware failure or file specification errors that does not apply elsewhere. The first of these points means that text saving commands have no use for index codes, and this provides the opportunity to deal with the second issue by specifying filenames in place of index codes, and thus maintaining a parallel syntax to other script commands. The third point impacts on the treatment of any actions that are associated with text saving commands: such actions are performed only if the relevant command is *successful*, and this provides a method for testing whether saving has actually taken place. (There is also a fourth, more subtle, difference between text saving and other commands, deriving from the first point above and affecting the interpretation of associated conditions. If other script commands are defined conditionally, the associated condition does not determine whether *the command itself* is carried out, but rather, specifies a persisting condition for when *the transformation (etc.) defined by the command* will be available. In the case of text saving commands, however, no other structure is defined by the command, and hence the condition is interpreted as applying directly to the text saving operation itself.)

Filenames and File Paths

In its most basic form, the `S` script command is as follows:

```
Smyfile.txt This text is to be saved
```

Note that the filename must occur immediately after the command, and cannot contain a space (for details of other prohibited characters, see §6.3, 'Alphabets and Special Symbols'). However if a space is required, this can be indicated by using an asterisk:

```
Sthis*is*my*file.txt This text is to be saved
```

Here the text will be saved into a file named '`this is my file.txt`', within the *Elizabeth* directory (i.e. the directory from which the program was started). Other directories can also be used, either by giving an absolute file path:

```
SC:\AI\Elizabeth\test\myfile.txt This text is to be saved
```

or a path specification relative to the *Elizabeth* directory:

```
Sscripts\test\fileone.txt This text is to be saved
S..\test\filetwo.txt This text is to be saved
```

If the *Elizabeth* directory is `C:\AI\Elizabeth`, then these last two commands will save to the files:

```
C:\AI\Elizabeth\scripts\test\fileone.txt
C:\AI\test\filetwo.txt
```

Note that the file path referred to is automatically created (if possible), so there is no problem with including many levels of directory structure within the file path specification.

Appending, Restarting, and Deleting Files

Standardly the `S` command either starts a new file and writes text to it, or appends the text to an existing one. Thus in the context of a transformation within which `[phrase]` is defined,

```
Smyfile.txt [phrase]
```

will create a new file if necessary, but if `myfile.txt` already exists (within the *Elizabeth* directory), then the text of `[phrase]` will be appended to the end of that file. If you wish to add an end-of-line to the file after the appended text, then use the special symbol '\n':

```
Smyfile.txt [phrase]\n
```

This special symbol can be used at any point in the text, repeatedly if required:

```
Smyfile.txt These\nwords\nare\non\ndifferent\nlines\n
```

To delete any previous file of the same name, and start from the beginning of a new one, simply add a backslash '\' immediately following the filename:

```
Smyfile.txt\ These\nwords\nare\non\ndifferent\nlines\n
```

so following this command, the file will contain only the text specified. There is no difference between appending and starting a new file if a file with the specified name does not already exist – in effect, the '\' after the filename just inserts a file deletion before the text saving command. Deleting a file is a logical extension of this, where the previous file of the same name (if any) is deleted, but no text is specified to start a new file:

```
Smyfile.txt\
```

Note that this can result in a file's being deleted unexpectedly if the command includes a pattern that matches nothing, for example:

```
Smyfile.txt\ [phrase?]
```

If in this situation (where `[phrase?]` is empty) you wish to create a zero-length file rather than no file at all, the solution is simply to use two commands which separate out the file deletion from the file writing:

```
Smyfile.txt\
Smyfile.txt [phrase?]
```

The latter command will start by creating a zero-length file if it does not already exist, even if there is then no text to be added to it.

6.1.6 Input, Output, and Final Transformations

The function of input, output, and final transformations is explained in §2.3, 'The Input/Keyword/Output/Final Transformation Process'. All of them have the same syntactic options, illustrated below using input transformations only:

`I one => two`	add the transformation 'one => two' to the set of input transformations (and if there is already a transformation of this form, replace it)
`I one =>`	add the transformation 'one =>' to the set of input transformations, i.e. the transformation which deletes the string 'one', by replacing it with the null string (etc.)
`I\ one => two`	delete the transformation 'one => two' from the set of input transformations (if it exists)
`I\`	delete all input transformations

Note that the command '`I`' by itself, or commands such as '`I => two`', will generate an error message, because it is not possible to have a null input transformation or one with a null left-hand side.

`I12 one => two`	add the transformation 'one => two' to the set of input transformations, giving it the index code '12' (and replacing any transformation that already has '12' as an index code)
`I12\`	delete the input transformation having the index code '12' (if there is one)
`I12\ one => two`	delete the input transformation 'one => two' which has the index code '12' (if there is such an input transformation)

As before, the difference between the last two of these is that the last makes two checks rather than one before deletion – it makes a deletion only if the set of input transformations contains a transformation which *both* has the form 'one => two' *and also* has the index code '12'. Single- and double-checking possibilities also apply to another type of deletion command, where the transformation to be deleted is identified by its left- or right-hand side only rather than the entire transformation:

`I\ one`	delete the first input transformation of the form 'one => ?' (i.e. whose left-hand side is 'one' but whose right-hand side can be anything); here as usual the search for the 'first' is performed in alphabetical order of index code
`I\ => two`	delete the first input transformation of the form '? => two' (i.e. whose right-hand side is 'two')

`I12\ => two`	delete the input transformation having the index code ‘12’ (if there is one), but only if it is of the form ‘? => two’.
`I12\ one`	delete the input transformation having the index code ‘12’ (if there is one), but only if it is of the form ‘one => ?’.

Note that this last example is different from the one below: without ‘=>‘, the right-hand side of the deleted transformation can be anything at all, but with ‘=>‘, a specific right-hand side (namely, the null replacement) is intended:

`I12\ one =>`	delete the input transformation having the index code ‘12’ (if there is one), but only if it is of the form ‘one =>’

Conditions and Actions

Input, output, and final transformations can all be defined as conditional in the standard way. So for example:

```
<[Mroman]>: I two => II
```

defines an input transformation which will apply only if the memory `[Mroman]` has been defined. If you define such a conditional transformation and later wish to delete or replace it, then you can repeat the condition in your deletion or replacement command to ensure that the right transformation is identified (though there is no need to include the condition unless there is a risk of ambiguity, and an index code usually provides a more convenient method of reidentification). So for example:

```
<[Mroman]>: I\ two => II
```

will make a deletion only if a transformation exists with the specified form and the same condition.

Actions associated with input, output, and final transformations have the obvious interpretation: they are performed whenever the transformation in question is used. So for example the transformation:

```
I MUM => MOTHER
  & {Minformal}
```

will save a relevant memory whenever the specified transformation is used. Note here that if two uses of a transformation are made *simultaneously* (e.g. where ‘MUM IS MY MUM’ is changed to ‘MOTHER IS MY MOTHER’ through the above input substitution), the associated action will be performed once only. In cases of iteration or cycling, however (see §5.2, ‘Iteration and Cycling of Input/Output/Final Transformations’), a transformation can be used many times in succession resulting in repeated actions, so care should be taken accordingly.

6.1.7 Keyword Phrases

The general operation of keywords and responses is explained in the §2.3, ‘The Input/Keyword/Output/Final Transformation Process’. However the syntactic options for keyword phrases and responses are significantly more complex than for any other command type, because the keywords fall into distinct sets, each associated with a set of responses – each of these pairs of sets has its own index code, in addition to the index codes for the individual keywords and responses. As well as these standard index codes, the system provides some simpler ways of operating on the most relevant sets; in particular, it keeps track of which set-pair was the last to be modified in some way and provides the special code ‘@’ to refer to it – here we call this set-pair the ‘current’ set. (For convenience, in what follows reference is often made to ‘the current keyword set’ or ‘the current response set’; note, however, that it is really the *pair* of sets which are current, and the two halves of the pair always go together – creation or deletion of either of them automatically creates or deletes the other.)

Keyword Addition (and Replacement)

The behaviour of the keyword addition commands can be affected by whether or not the current set is ‘response-free’ (i.e. has not yet been associated with any responses). When keywords and associated responses are to be read in from a simple Script (like the Illustrative Script of §1.4), the keywords are grouped together in the Script file, followed directly by the associated responses. To accommodate this standard layout, the default mechanism for reading keywords works as follows. While the current set’s keywords are being read in (i.e. as long as that set remains response-free), any further keywords will be put into that current set. But once some associated responses have been added (i.e. after the current set has ceased to be response-free), the next keyword will

instead trigger the creation of a new response-free current set, into which that keyword will be put. This behaviour is summarised here:

`K SURE`	Add the keyword 'SURE' to the current set if that set is response-free; otherwise create a new (and hence response-free) current set, and add the keyword 'SURE' to this new set

Note that a keyword-free current set will be treated in the same way as a response-free set (whether or not it contains any responses), though this situation is unlikely to arise when a Script is being read, because usually a set is created only when the first keyword is being added to it.

Note also that the behaviour summarised above does not apply to dynamic commands (as opposed to script commands) in situations where the specified keyword already exists. In dynamic contexts, the command given first checks the current set and then any other keyword sets to see whether 'SURE' is already a defined keyword – if so, it is replaced within the set concerned (which usually leaves the system's behaviour unaffected, but which (a) avoids the creation of unwanted duplicates in dynamic scripts; (b) can change the system's behaviour if the keyword has associated actions; and (c) provides a way of making the relevant keyword set 'current' without any need for index codes).

The symbol '@' can be used in any keyword or response command to refer to the current set, and this can be used to override the default behaviour above:

`K@ SURE`	add the keyword 'SURE' to the current set, whether or not that set is response-free (but if no set is current, first create a new empty keyword set and make it current)

An alternative is to use a set index code:

`Ksi SURE`	add the keyword 'SURE' to the set whose index code is 'si' (which then becomes the current set); if such a set does not already exist, then create it before adding the new keyword

If the set concerned already contains the given keyword (i.e. 'SURE' in these cases), then the keyword command will result in a direct replacement (retaining the same keyword index code, but possibly modifying associated actions) rather than an addition. When such replacement is definitely intended, however, you may wish to search through all the existing keyword sets looking for a match, rather than simply looking in the current or some other named set:

`K/ SURE`	search through all the existing keyword sets for the keyword 'SURE' – if found, directly replace it with the new keyword 'SURE' (and associated actions etc.); if not found, do nothing

If you already know which set you wish to search, then you can use one of the following:

`K@/ SURE`	search through the current keyword set for the keyword 'SURE' – if found, directly replace it with the new keyword 'SURE' (and associated actions etc.); if not found, do nothing
`Ksi/ SURE`	search through the keyword set whose index is 'si' for the keyword 'SURE' – if found, directly replace it with the new keyword 'SURE' (and associated actions etc.) and make 'si' the current set; if not found, do nothing

You may occasionally wish to create a new empty keyword set (e.g. so that keywords can then be added to it as the 'current' set). This can be done by using similar commands to those above, but without any keyword phrase:

`K`	create a new empty keyword set, and make it the current set
`K@`	*if* there is no current keyword set, then create a new empty keyword set and make it the current set
`Ksi`	create a new empty keyword set under the index code 'si' (unless a set with that code already exists), and make it the current set

As mentioned above, individual keywords have their own index codes, and these can be used either by themselves, or combined with set index codes (note here that a keyword index is always preceded by a '/'):

`K@/ki SURE`	add the keyword ‘SURE’ to the current set, with the keyword index ‘ki’; if there is no current set, create one before adding the keyword
`Ksi/ki SURE`	add the keyword ‘SURE’, with the keyword index ‘ki’, to the set whose index code is ‘si’; if such a set does not already exist, then create it before adding the new keyword
`K/ki SURE`	search through all the existing keyword sets for the keyword ‘SURE’ having index code ‘ki’ – if found, directly replace it with the new keyword ‘SURE’ (and associated actions etc.); if not found, do nothing

Keyword Deletion

The simplest deletion command is a special case, designed to provide the behaviour which is most likely to be needed in a straightforward Script that makes no use of index codes:

`K\ SURE`	delete the keyword ‘SURE’ from the current set if it is present there; otherwise, search through all the existing keyword sets for the keyword ‘SURE’, and if found, delete it (but note that this will only delete the *first* found instance); the set affected, if any, then becomes the current set

The other keyword deletion commands are more systematic, and most are direct complements of the keyword addition commands above, using the deletion symbol ‘\’ immediately after the ‘K’ and any index codes:

`K@\ SURE`	delete the keyword ‘SURE’ from the current set, if it is present there
`K@/\ SURE`	delete the keyword ‘SURE’ from the current set, if it is present there, and if this leaves the set empty of keywords and responses, delete the entire set
`Ksi\ SURE`	delete the keyword ‘SURE’ from the set whose index code is ‘si’, if it is present there
`Ksi/\ SURE`	delete the keyword ‘SURE’ from the set whose index code is ‘si’, if it is present there, and if this leaves the set empty of keywords and responses, delete the entire set
`K/\ SURE`	search through all the existing keyword sets (in alphabetical order of set index codes) for the keyword ‘SURE’ – if found, delete it (but note that this will only delete the *first* found instance)
`K@/ki\ SURE`	delete the keyword ‘SURE’, with the keyword index ‘ki’, from the current set (assuming that such a keyword is in that set)
`Ksi/ki\ SURE`	delete the keyword ‘SURE’, with the keyword index ‘ki’, from the set whose index code is ‘si’ (assuming that such a keyword is in that set)
`K/ki\ SURE`	search through all the existing keyword sets for the keyword ‘SURE’ having index code ‘ki’ – if found, delete it (but note that this will only delete the *first* found instance)

If you make use of keyword index codes, then when deleting keywords you are most likely to use the relevant codes only, rather than repeating the keyword phrases themselves:

`K@/ki\`	delete the keyword with index ‘ki’ from the current set (assuming that such a keyword is in that set)
`Ksi/ki\`	delete the keyword with index ‘ki’ from the set whose index code is ‘si’ (assuming that such a keyword is in that set)
`K/ki\`	search through all the existing keyword sets for a keyword having index code ‘ki’ – if found, delete it (but note that this will only delete the *first* found instance)

When deleting entire sets of keywords, you can normally choose either to leave an empty set of keywords (in which case the associated set of responses will be unaffected), or to remove the set entirely (together with the associated set of responses). The difference is signified by inclusion of ‘/’ before the final ‘\’:

`K\`	delete all keywords, leaving all keyword sets empty (but still in existence)
`K/\`	delete all keyword and response sets

`K@\`	delete all keywords from the current set, leaving it empty (but still in existence)
`K@/\`	delete the current keyword and response sets
`Ksi\`	delete all keywords from the set having index code 'si', leaving it empty (but still in existence) and making it the current set
`Ksi/\`	delete the keyword and response sets which have index code 'si'

6.1.8 Keyword Responses

The commands for keyword responses are very similar to those for keyword phrases, and perhaps the only really significant difference concerns the first and simplest command listed below:

Response Addition (and Replacement)

The basic response addition command is simpler than its keyword equivalent; note that whereas '`K PHRASE`' will often involve the creation of a new keyword set (and corresponding response set), '`R PHRASE`' always concerns only the current set if there is any:

`R SURE?`	add the response 'SURE?' to the current set (but if there is no current set, first create a new empty response set and make it current)

This implies identical behaviour to the command:

`R@ SURE?`	add the response 'SURE?' to the current set (but if there is no current set, first create a new empty response set and make it current)

Just as with keyword phrases, other sets can be accessed using set index codes:

`Rsi SURE?`	add the response 'SURE?' to the set whose index code is 'si' (which then becomes the current set); if such a set does not already exist, then create it before adding the new response

If the set concerned already contains the given response (i.e. 'SURE?' in these three cases), then the response command will result in a direct replacement (retaining the same index code, but possibly modifying associated actions) rather than an addition. When such replacement is definitely intended, however, you may wish to search through all the existing response sets looking for a match, rather than simply looking in the current or some other named set:

`R/ SURE?`	search through all the existing response sets for the response 'SURE?' – if found, directly replace it with the new response 'SURE?' (and associated actions etc.)

You may occasionally wish to create a new empty response set (e.g. so that responses can then be added to it as the 'current' set). This can be done by using similar commands to those above, but without any response phrase. Since creation of a keyword set always goes together with the creation of a corresponding response set, however, these three commands are identical in effect with '`K`', '`K@`' and '`Ksi`' respectively:

`R`	create a new empty response set, and make it the current set
`R@`	*if* there is no current response set, then create a new empty response set and make it the current set
`Rsi`	create a new empty response set under the index code 'si', and make it the current set

As mentioned above, individual responses have their own index codes, and these can be used either by themselves, or combined with set index codes (note that a response index is always preceded by a '/'):

`R@/ri SURE?`	add the response 'SURE?' to the current set, with the response index 'ri'
`Rsi/ri SURE?`	add the response 'SURE?', with the response index 'ri', to the set whose index code is 'si'; if such a set does not already exist, then create it before adding the new response
`R/ri SURE?`	search through all the existing response sets for the response 'SURE?' having index code 'ri' – if found, directly replace it with the new response 'SURE?' (and associated actions etc.)

Response Deletion

Just as with the basic response addition command, so the basic response deletion command is simpler than its keyword equivalent (because it does not involve any search through other sets if the response to be deleted is absent from the current set – this is because the same response can often occur in more than one set, whereas having the same keyword in more than one set is usually pointless):

`R\ SURE?`	delete the response 'SURE?' from the current set if it is present there

The other response deletion commands are more systematic, and most are direct complements of the response addition commands above, using the deletion symbol '\' immediately after the 'R' and any index codes:

`R@\ SURE?`	delete the response 'SURE?' from the current set, if it is present there
`R@/\ SURE?`	delete the response 'SURE' from the current set, if it is present there, and if this leaves the set empty of keywords and responses, delete the entire set
`Rsi\ SURE?`	delete the response 'SURE?' from the set whose index code is 'si', if it is present there
`Rsi/\ SURE?`	delete the response 'SURE' from the set whose index code is 'si', if it is present there, and if this leaves the set empty of keywords and responses, delete the entire set
`R/\ SURE?`	search through all the existing response sets (in alphabetical order of set index codes) for the response 'SURE?' – if found, delete it (but note that this will only delete the *first* found instance)
`R@/ri\ SURE?`	delete the response 'SURE?', with the response index 'ri', from the current set (assuming that such a response is in that set)
`Rsi/ri\ SURE?`	delete the response 'SURE?', with the response index 'ri', from the set whose index code is 'si' (assuming that such a response is in that set)
`R/ri\ SURE?`	search through all the existing response sets for the response 'SURE?' having index code 'ri' – if found, delete it (but note that this will only delete the *first* found instance)

If you make use of response index codes, then when deleting responses you are most likely to use the relevant codes only, rather than repeating the response phrases themselves:

`R@/ri\`	delete the response with index 'ri' from the current set (assuming that such a response is in that set)
`Rsi/ri\`	delete the response with index 'ri' from the set whose index code is 'si' (assuming that such a response is in that set)
`R/ri\`	search through all the existing response sets for a response having index code 'ri' – if found, delete it (but note that this will only delete the *first* found instance)

To delete entire sets of responses, use the following commands:

`R\`	delete all responses, leaving all response sets empty (but still in existence)
`R@\`	delete all responses from the current set, leaving it empty (but still in existence)
`Rsi\`	delete all responses from the current set, leaving it empty (but still in existence)

To remove the keyword/response set pairs entirely, use the corresponding '`K`' commands (for which see the previous section). To preserve the parallel with '`K`' commands, the alternative forms '`R/\`', '`R@/\`', and '`Rsi/\`' respectively are also permissible, but the '`K`' forms are to be preferred.

6.1.9 Keywords and Responses Involving Conditions and Actions

Both keywords and responses can be defined as conditional in the standard way, for example:

```
<[Mmem]==ok>: K SURE
```

which will treat 'SURE' as a keyword only while the memorised phrase `[Mmem]` is defined and has the value 'ok'.

If you define such a conditional keyword or response and later wish to delete or replace it, then you can repeat the condition in your deletion or replacement command to ensure that the right keyword or response is identified (though there is no need to include the condition unless there is a risk of ambiguity, and an index code usually provides a more convenient method of reidentification). So for example:

```
<[Mmem]==ok>: K@\ SURE
```

will delete the keyword 'SURE' from the current set, if it is present there and is associated with the condition `<[Mmem]==ok>`.

Actions associated with keywords or responses have the obvious interpretation: they are performed whenever the keyword or response in question is used. So for example the response:

```
R MY NAME IS FRED
  & {Mmyname FRED}
```

will save an appropriate memory when it is used. Note that the action will only be performed if the keyword or response in question is *used* – this will not apply, for example, where a keyword (with an associated action) is initially recognised but cannot be used because all the responses in the same set have conditions that are not satisfied.

6.2 Built-In String Functions

The system provides some string functions that can be used to adjust the output of any transformation. The names of these functions themselves can be used in either upper- or lower-case, according to your own preference (and you can also choose whether or not to include spaces after the colon).

6.2.1 The Case Functions `[UPPER:]` and `[LOWER:]`

Occasionally, you may wish to change part of all of the active text, or text that is being substituted with some transformation, from upper to lower case or vice-versa. To do this, simply use the functions `[LOWER:]` or `[UPPER:]` respectively within an appropriate transformation command. For example:

```
I [X] => [lower:[X]]
```

can be used to change the entire active text to lower case at some point within the input transformation processing stage. Likewise:

```
O [letter][word?] => [upper:[letter]][lower:[word?]]
```

can be used during the output processing stage to capitalise the first letter of each word, and make the rest lower case.

6.2.2 The Increment/Decrement Functions `[INC:]` and `[DEC:]`

The functions `[INC:]` and `[DEC:]` respectively increment and decrement numerical values, as follows:

```
[INC: [term]]
```

will give 'ABC1' if `[term]` is 'ABC', 'XYZ5' if `[term]` is 'XYZ4', '100' if `[term]` is '99', etc. – the rule here is simply to increment the final number if the parameter ends with one or more digits, and add a final '1' otherwise;

```
[DEC: [term]]
```

will give 'ABC-1' if `[term]` is 'ABC' or 'ABC0', 'XYZ3' if `[term]` is 'XYZ4', '99' if `[term]` is '100', etc. – the rule here is to decrement the final number if the parameter ends with one or more digits, and add a final '-1' otherwise.

These functions can be used to implement arithmetical operations as explained in §3.3, 'The Power of Recursion', and also to automate changes of 'state' whereby one memory is replaced by another. For an example of the latter usage, see §4.6, 'Implementing Propositional Logic'.

6.3 Alphabets and Special Symbols

When specifying transformations within the system, it can be helpful to know that certain characters are always 'filtered out' of the typed input – by using these characters within the transformation rules, one can be totally confident that the typed input will never cause conflicts with them (imagine, for example, what problems might arise if the characters '[' and ']' were permitted, so that the typed input could include strings such as '[phrase]'). Moreover when making use of index codes, it can be important to know the 'ASCII' ordering of the relevant characters, since this determines how they will be sorted and hence the order in which they will be processed.

The filtering mechanisms operate as follows, with '0…9', 'A…Z', and 'a…z' indicating the digits, upper-case letters and lower-case letters respectively (the space character is not shown, but is permitted throughout except within index codes; note however that multiple spaces are always reduced to single spaces). In general, characters not permitted at any stage are silently removed at the earliest possible stage in any processing. *(The only exceptions to these are the ASCII codes 145-8, representing the various 'curved' quotation marks, which are silently converted into their 'straight' equivalents.)*

Script lines and stored actions can contain the characters (in ASCII order):

! " $ % & ' ( ) * + , - . / 0…9 : ; < = > ? @ A…Z [\ ] ^ _ a…z { } ~

In addition, the character '#' can be used as the start of an '#INCLUDE' directive only, and '|' can be used to simulate the effect of a line break when inputting script commands from the keyboard using **F1** *or the File.menu facility.*

User input can contain the characters:

! " ' ( ) + , - . 0…9 : ; < > ? A…Z a…z

Input/output/final/keyword transformations and fixed responses can involve the characters:

! " $ % ' ( ) * + , - . 0…9 : ; < > ? @ A…Z [\ ] ^ _ a…z { } ~

(Though the specification of any input/output/final transformation must include the characters '=>', these are not stored as part of the transformation.)

Memorised phrases can include the characters:

! " $ % ' ( ) * + , - . 0…9 : ; < > ? A…Z ^ _ a…z ~

Index codes (of all kinds) can include the characters:

! " $ % ' ( ) * + , - . 0…9 : ; < > ? A…Z ^ _ a…z ~ *(spaces are <u>not</u> permitted within index codes)*

(Note that the forward and backslash characters '/' and '\' have specific meanings when combined with index codes within commands: '/' is used to separate the index code of a keyword set from that of a keyword or response within that set; '\' signifies a deletion rather than an addition command – hence attempted use of these characters within an index code can give rise to particularly confusing results. Note also that the character '@' can be used to represent the index code of the last-modified keyword set in keyword/response commands.)

File paths in 'S' commands can include the characters:

! # $ % & ' ( ) + , - . / 0…9 : ; = @ A…Z [\ ] ^ _ ` a…z { } ~

(Spaces themselves cannot be used here, but the asterisk ***** *– which is not itself permitted within file paths – can be used to indicate a space: i.e. all asterisks are treated as spaces in performing any 'S' command. Also note that the colon, slash and backslash characters are illegal in filenames, but permitted in file paths.)*

Program output can contain the characters:

! " ' ( ) + , - . 0…9 : ; < > ? A…Z a…z

Program output must end with one of the characters:

! . ?

6.4 Directory Usage, System Files, and Predefined Scripts

The main program, `Elizabeth.exe`, can be run from any directory on the PC, and this directory can be set as 'read only' (e.g. on a school network) to ensure that the software is not corrupted.

In the default setup, the current documentation file, `Elizabeth.pdf`, should also be present in this *Elizabeth* program directory, to enable it to be displayed from the first item ('Open Help file') in the Help menu. There should also be two subdirectories beneath the *Elizabeth* program directory, one named 'Illustrative Scripts' (acting as a library of standard scripts, and which can accordingly be set as 'read only'), and the other named 'My Scripts' (which must be readable and writable by the user). The latter is the directory used for user scripts, and if it does not exist initially, the program will attempt to create it.

However this default setup can be modified if a text file `Elizabeth.path` is present in the program directory. This can specify up to six different paths and filenames, as follows (note here that anything following ' /' in the file is treated as a comment, and ignored):

```
userpath=C:\Elizabeth         / Base path for user directories
userdir=My Scripts            / Default user scripts subdirectory, within base path
library=Illustrative Scripts  / Library subdirectory of program directory
startscript=Elizabeth.txt     / Script to load when starting up Elizabeth program
backupscript=EOriginal.txt    / Backup of startscript, within library subdirectory
pdfname=Elizabeth.pdf         / Name of Help file PDF within program directory
```

In practice, however, this file is likely to contain only a single line, to enable the *userpath* – which by default points to the program directory – to be set to a quite different directory. Again, this can be helpful if you wish to confine users to their personal area of a network while providing the software centrally. (The program attempts to create the entire directory path *userpath**userdir* if it does not already exist, so there is no need to set this up in advance, assuming that the appropriate permissions will be available to the user when running *Elizabeth*.)

When the program starts up, it attempts to load the script `Elizabeth.txt` (or the *startscript*, if another has been specified) from the 'My Scripts' (*userpath**userdir*) subdirectory. If that script file does not exist, then the program attempts to recreate it by copying the file `EOriginal.txt` (*backupscript*) from the 'Illustrative Scripts' (*library*) subdirectory into 'My Scripts', and renaming the copy `Elizabeth.txt`. This behaviour means that `Elizabeth.txt` can be edited freely, and eventually just deleted, without losing the original version (which contains the illustrative script of §1.4).

The files provided with the system are as follows. If installed correctly the first three should be in the chosen root directory, the fourth – `Elizabeth.txt` – in the 'My Scripts' subdirectory, and the remainder (all but one with the extension `txt`, and mostly listed here in order of relevant sections within this document) in the 'Illustrative Scripts' subdirectory. When one of these illustrative scripts is selected from the Help menu, the relevant file is automatically copied from 'Illustrative Scripts' into 'My Scripts' and opened within the latter subdirectory. Again, this means that it can be freely edited by the user, without corrupting the original copy.

`Elizabeth.exe` — The *Elizabeth* program – to run the program, execute this file (e.g. by clicking on it within *Windows* explorer).

`Elizabeth.pdf` — The PDF file that you are now reading, which is automatically opened when you select an appropriate option from *Elizabeth's* Help menu.

`Elizabeth.ppt` — A *PowerPoint* presentation on *Elizabeth*, leading up to the theory of grammars and resolution theorem proving in propositional logic.

`Elizabeth.txt` — The default Script file for *Elizabeth* – automatically loaded into the program (from the 'My Scripts' directory) when it starts, and designed to be 'played with' while reading the Illustrative Script and Conversation of §1.4, where it is explained. By editing this file, you can make the system start with whatever Script you choose (and note that the original version of this Script is preserved in the file `EOriginal.txt` in the 'Illustrative Scripts' subdirectory, so there is no risk of losing it through such editing).

`EOriginal.txt` — A backup copy of the original default Script file `Elizabeth.txt`. When the system starts up, if it cannot find the file `Elizabeth.txt` in the 'My Scripts' directory, then it will attempt to copy this file to recreate `Elizabeth.txt` before opening.

`EOrigMem.txt`	A modified version of the default Script file `Elizabeth.txt`, incorporating the simple memory and recall facility described in §2.4, 'Phrase Memorisation and Recall'.
`EOrigSplit.txt`	A modified version of the default Script file `Elizabeth.txt`, incorporating both the simple memory and recall facility described in the §2.4, and also the facility for dealing with multiple sentences described in §3.2, 'Recursion and Text Splitting'.
`ArithCycle.txt`	The implementation of arithmetic using input transformation cycling explained in §3.3.2, 'Using Recursion and Cycling for Arithmetic'.
`ArithRec.txt`	The implementation of arithmetic using recursion explained in §3.3.2.
`Arrays.txt`	Simple use of memory arrays explained in §2.4.2, 'Implementing Memory Arrays'.
`Hanoi.txt`	Solution to the 'Towers of Hanoi' puzzle explained in §3.3, 'The Power of Recursion'.
`Passive.txt`	The transformation from active to passive verbs explained in §4.1, 'Implementing Grammatical Rules'.
`Questionr.txt`	The questionnaire example explained in §4.5, 'Defining and Using Conditional Commands'.
`Resolution.txt`	The implementation of resolution theorem proving for propositional logic explained in §4.6, 'Implementing Propositional Logic'.
`TagQuest.txt`	The 'tag questions' example explained in §4.1, 'Implementing Grammatical Rules'.
`TurboAll.txt`	A Script based on the freeware ELIZA program 'Turbo Eliza' mentioned in §2.1, 'Introduction to Script Commands', and provided here for illustration and experimentation.
`TurboEliza.txt`	The main file of a modularised version of the same 'Turbo Eliza' Script. The module files which it incorporates using '#INCLUDE' directives (see §2.1) are as follows:
`TurboFixed.txt`	The 'Turbo Eliza' *fixed responses* (i.e. welcome/quitting messages, and void input/no-keyword responses – see §2.2, 'Simple Message Selection Commands', for explanation of how these work). To incorporate these within your own Script, use the directive: `#INCLUDE TurboFixed.txt`
`TurboMy.txt`	The 'Turbo Eliza' module which enables 'my' phrases to be repeated back to the user (e.g. so the input 'My car won't work' can later result in the output 'DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR CAR WON'T WORK?'). For explanation see §2.4, 'Phrase Memorisation and Recall', and to incorporate these commands within your own Script, use the directive: `#INCLUDE TurboMy.txt`
`TurboInput.txt`	The 'Turbo Eliza' *input transformations* (see §2.3) – to incorporate within your own Script, use the directive: `#INCLUDE TurboInput.txt`
`TurboKeys.txt`	The 'Turbo Eliza' *keyword transformations* (see §2.3) – to incorporate within your own Script, use the directive: `#INCLUDE TurboKeys.txt`
`TurboOutput.txt`	The 'Turbo Eliza' *output transformations* (see §2.3) – to incorporate within your own Script, use the directive: `#INCLUDE TurboOutput.txt`
`TuringGCD.txt`	Implementation of a Turing machine to calculate the greatest common divisor of two numbers, as discussed in §4.7, 'Implementing a Turing Machine'.
`Weizenbaum.txt`	A faithful translation of Joseph Weizenbaum's DOCTOR script (see §1.5), the basis of the famous original ELIZA program as published in January 1966.
`WeizDoctor.doc`	The dialogue originally published by Joseph Weizenbaum in January 1966 and reproduced in §1.5. This can be 'replayed' within the system using the facilities provided in the File menu (see §1.2.1).

6.5 Script Directives

Most of the available script directives provide means of enforcing menu options when a script is run, which will temporarily override the defaults set by the menus themselves. While a script is running, the operative settings can be seen using the Current Settings Panel (accessed through the View menu – see §1.2.2). These directives fall into three groups, corresponding to the three relevant menus, as explained in the following sections. The system also provides for '#INCLUDE' directives, as explained in §2.1.2, 'Script Layout and Modularisation'.

6.5.1 View Menu Settings: /V Directives

Here is a list of the various /V directives that are recognised by the system, corresponding straightforwardly with options available through the View menu and affecting the 'look and feel' of the system:

```
/V Show all tables
/V Show transformation tables
/V Hide tables
/V Instant response
/V Fast typing
/V Medium typing
/V Slow typing
```

6.5.2 Processing Menu Settings: /P Directives

Here is a list of the various /P directives that are recognised by the system, corresponding straightforwardly with options available through the Processing menu, and affecting how inputs are processed into outputs:

```
/P Final punctuation ON
/P Final punctuation OFF
/P Sequential responses
/P Randomised responses
/P Echo if no keywords
/P Blank if no keywords
/P Upper case output
/P Lower case permitted
```

6.5.3 Control Menu Settings: /C Directives

Here is a list of the various /C directives that are recognised by the system, corresponding with options available through the Control menu and the Recursion/Iteration/Cycling Control Panel, which affect recursion, and the repetition and sequencing of input, output, and final transformations. Wherever a number is shown (i.e. '5000', '500', or '10'), this indicates the system default, and can be replaced in a directive by any other number.

```
/C Recursion ON
/C Recursion OFF
/C Matchlimit 5000
/C Memchecklimit 500
```

Each of the following directives comes in three forms, which are illustrated explicitly in the case of Input Iteration, but subsequently abbreviated:

```
/C Input Iteration unlimited
/C Input Iteration prevented
/C Input Iteration undo limit: 10

/C Output Iteration unlimited/prevented/undo limit: 10
/C Final Iteration unlimited/prevented/undo limit: 10
/C Input Cycling unlimited/prevented/undo limit: 10
/C Output Cycling unlimited/prevented/undo limit: 10
/C Final Cycling unlimited/prevented/undo limit: 10
```

6.6 Copyright, and Versions of the Software

The *Elizabeth* software and accompanying documentation are the copyright of Peter Millican (c) 2002-18, but are provided free for educational and recreational use, provided that the system files (including this document and all of the other files listed in the section on 'Directory Usage, System Files, and Predefined Scripts' (§6.4), with the sole exception of `Elizabeth.txt`) remain unmodified and are retained together. Anyone wishing to use *Elizabeth* for any commercial purpose should contact the author at the email address given above.

This is Version 2.20 of the software, released in July 2018. It revises the previous menu organisation, to reflect more systematically the logic of the various settings (now in the View and Processing menus, and the RIC Control Panel), and introduces the Current Settings Panel, so that settings set from scripts can be tracked without disrupting the menu defaults. It also provides more flexible configuration using the Elizabeth.path configuration file, as explained in §6.4. Version 2.10, released in June 2018, finally bowed to the inevitable in removing any dependence on Microsoft's now defunct (but previously very useful) "Help" facility, resulting in the current PDF documentation and corresponding modifications to the Help menu operation. Apart from that, it corrected a bug in saving settings from the Control menu, and introduced the *memory checking limit*, so as to avoid circular dependence in memory definitions.

The main change in Version 2.07 of the software, released in May 2014, brought some additions to the help documentation and also adaptations to allow the software to run on a system in which both the program's directory and the 'Illustrative Scripts' subdirectory are 'write-protected' so that users are unable to save or edit files there – this is likely to be desirable in a classroom situation. To achieve this, a new subdirectory 'My Scripts' is automatically created, and the operation of the various loading and saving operations has been adapted accordingly (including copying of the illustrative scripts, when they are selected).

Version 2.06 of the software, released in May 2013, together with 2.05 (January 2010, but not released publicly) provided the following main new features:

- Enhanced capacity (up to 1000 words in each input; up to 100 pending actions; up to 100 responses in each fixed set; up to 5000 input/output/final transformations and keysets; up to 100 keywords and responses in each set; up to 5000 memories, memorised inputs/outputs, and lines in saved dialogue);
- Memory indirection (e.g. '[M[Mindex]]') to allow array manipulation (as demonstrated in a new illustrative script);
- Multiple tracing options in Analysis menu, including 'Deep match tracing' to see detailed processing;
- Menu option to have responses selected sequentially or in random order (avoiding immediate repetition) – either option can also be specified for particular sets (e.g. 'K!' or 'V!' to make a keyword set or void input responses sequential, 'K?' or 'V?' to make them random.
- Possibility of adding e.g. '{[word]=>K003}' to start recursion starting from keyword set 003; likewise for input, output and final transforms;
- Void input and no keyword responses moved prior to final transformations, to enable final processing of them;
- '_' allowed as part of a word, e.g. to enable a [phrase] to include special terms for categories;
- Option added to suppress echoing of input, to facilitate handling of separate recursion;
- Facility to open and take inputs from a previously saved dialogue (rather than having to remember them);
- 'Doctor' script, algorithmically equivalent to that listed on pp. 44-5 of Joseph Weizenbaum's original article (*Communications of the ACM*, January 1966, pp. 36-45), together with the original published dialogue;
- Illustrative script demonstrating implementation of a Turing machine.

Version 2.05 corrected a problem with the dialogue box sizing under versions of Windows after XP, and 2.06 corrected a small bug whereby a previous loaded Welcome message would be used if a new script was loaded that had none. In the previous release 2.04 (February 2006), the most significant additions were:

- A facility for saving text to files (as described in the new section on 'Saving Text Files Dynamically', §2.5);
- 'Exit messages' to provide more control of exiting and quitting, enabling a script to determine when these are permitted (see §2.2 on 'Simple Message Selection Commands' for details);
- The possibility of defining messages, transformations, keywords and responses as *self-deleting*, so that once used they will automatically be removed from the system, usually to avoid repetition (see 'Self-Deletion' in §4.2, 'Dynamic Script Processing');

- A new pattern, `[!]`, which can match any sequence of punctuation (see §3.1 on 'Pattern Matching');
- Enhanced treatment of conditions by allowing a final '?' to signify that the condition in question should be counted as *true* rather than *false* if memories used within it do not exist; also more systematic treatment of conditions, as reflected in the significant additions to the Command Syntax Reference Guide, §6.1.

Version 2.04 also corrected two small bugs: one which prevented actions' being associated with a Welcome message, and another that could in certain circumstances prevent a [?] pattern matching against nothing (associated with an error in the questionnaire illustrative script which did not match with the documented version). Version 2.03, released in February 2005, adjusted some error messages and improved the treatment of illustrative scripts, enabling these to be stored permanently in a subdirectory and copied when required. Version 2.02, released in January 2004, added the 'pause' debugging facility (explained in the section on Recursion and Text Splitting, §3.2) and corrected a couple of very minor bugs in version 2.01 from March 2003 (namely, the centring of information messages and query dialogues, and automatic recreation of `Elizabeth.txt` from `EOriginal.txt`). Changes in version 2.01 were rather more significant, including improvements to the Help menu options, the editor find and replace facility, and the bracket matching routine; also the addition of printer dialogues and 'safety checks' to avoid losing editing changes by quitting, and changes to the grammar rule and questionnaire scripts. The documentation was at that point supplemented with teaching (or self-learning) materials in the form of a 72-slide *PowerPoint* presentation based on part of the author's lecture series *Introduction to Artificial Intelligence* given to first year Computing students at Leeds University in 2002-03, and, in §1.6, an adapted version of the exercises provided for that course (with thanks to Dr John Stell).

Version 2.0 of the software, released in January 2003, incorporated a number of very significant improvements over previous versions. Scripts produced on earlier versions cannot be guaranteed to operate correctly on the new system, because some of the improvements made to increase its power and facilitate future development have involved changes to the command syntax and (especially) its pattern-matching routines. Future upgrades are intended to retain backward compatibility with the new Script syntax, as well as possible further improvements such as:

- Built-in functions for alphabetical and numerical comparison, and for the length of a string.
- Automated processing of files, as well as of keyboard input, so that the system can be used as a versatile file filter or processor, as well as a conversation system.
- Ability to partition transformations into distinct sets, for more sophisticated recursive behaviour.
- High-level processing control, to enable the processing sequence to be varied dynamically.
- Possibility of assigning 'activation weights' to keyword/response sets, to enable them to be prioritised dynamically depending on matches with a defined 'activation string'. This could be a powerful aid in creating 'directional' conversations such as those associated with Colby's famous PARRY program.
- Automatic calculation of a 'fog index', to facilitate the definition of behaviour that depends on the complexity of the input.
- Boolean combinations within conditions, enabling complex conditions to be defined relatively straightforwardly (rather than having to construct them one element at a time).

Any other suggestions for developments that might enhance the power and versatility of the system, especially for an educational context, will be greatly appreciated.

Peter Millican, peter.millican@hertford.ox.ac.uk