Using the *Elizabeth* Conversation Program

1

Using *Elizabeth*

An introduction to Chatbots
and Natural Language Processing

Peter Millican, July 2018
Hertford College, University of Oxford

1

2

Obtaining the Software

- The *Elizabeth* software can be downloaded from the *Elizabeth* home page at:
  www.philocomp.net/ai/elizabeth
  - The system is available either as a 'zip' file or as a 'self-extracting zip' file. Whichever you choose, unpack the files onto the machine you are using, e.g. into the folder C:\Elizabeth on your own machine, or C:\Temp\Elizabeth if you do not have permission to create a new folder.
- For links to other chatbot systems (as well as to *Elizabeth*), see www.simonlaven.com

2

3

Running the Software

- The main system file is called:
  Elizabeth.exe
- To run it, simply identify this file within *Windows Explorer*, and double-click.
- Make sure that the help documentation file:
  Elizabeth.pdf
  is in that same directory (the system also uses "My Scripts" and "Illustrative Scripts" subdirectories).
- Use the first item in *Elizabeth*'s Help menu to view the contents of the help file, and start reading – everything you need to know about the system is explained there! (You can also use the Help menu to load various illustrative scripts.)

3

4

Recent Versions of *Elizabeth*

- The latest version of *Elizabeth* on the website is 2.20. This has been adapted (in 2018) to remove dependence on the old *Windows* Help system (unavailable on *Windows* 10), so the Help documentation is now in the PDF file *Elizabeth.pdf*. Section references in this presentation (e.g. " § 1.4") are to that PDF document.
- The previous main version, 2.07, provided significant improvements in power (thus enabling a Turing machine simulator and original ELIZA clone), e.g.:
  - Memory indirection to allow creation of arrays;
  - Keyword sets can be either sequential or randomised;
  - Support for arbitrary recursion and text splitting/combining;
  - Trace enhancement with multiple options.

4

5

Playing Around

- *Elizabeth's* behaviour is based on a 'Script' file.
- Initially, *Elizabeth* should start up with a Script which shows the 'Welcome' message:

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

  If this doesn't happen for any reason, try locating and loading the script file *Elizabeth.txt* using 'Load script file and start' from the File menu.
- To familiarise yourself with *Elizabeth*, just play around a bit, typing input sentences, clicking on 'Enter', and seeing what happens. Note that the conversation is recorded in the 'Dialogue' tab.

5

6

The Illustrative Conversation

- Take a look at the section 'Illustrative Script and Conversation', § 1.4 in *Elizabeth.pdf*. Try typing inputs similar in style to what are shown in the part headed 'The Conversation' (§ 1.4.2), e.g. type in sentences containing words or phrases such as:
  - 'mum' or 'dad'
  - 'I think …'
  - '… is younger than …'
  - 'I like …ing'
- While doing this, look at the 'Trace' tab (just right of the 'Dialogue' tab): this shows how your input is being processed to produce the system's replies.

6

Using the *Elizabeth* Conversation Program

7

The Script Editor

- From *Elizabeth's* File menu, select 'Transfer script into Script Editor' – this will start up the Script Editor with the current script file loaded in.
- Now make a change to the 'Welcome' message (appears after 'W' in the second line of the Script); then from the *Editor's* File menu, select 'Restart Elizabeth after saving' – this will save your change, and restart *Elizabeth* using this edited script file (with its new 'Welcome' message).
- Note that the two File menus give various options for switching between *Elizabeth* and the *Editor*.

7

8

The First Illustrative Script

- Try to work out how the Script that you see within the *Editor* is determining *Elizabeth's* conversational behaviour – if any of it seems puzzling, refer to 'Illustrative Script and Conversation' (§ 1.4).
- Try playing around with the Script (like you did already with the 'Welcome' message), and see what effect this has on *Elizabeth's* conversation.
- Carry on doing this as we now explore *Elizabeth's* data tables as shown in the various system 'tabs'. Most of what you see in the tables comes directly from the Script file.

8

9

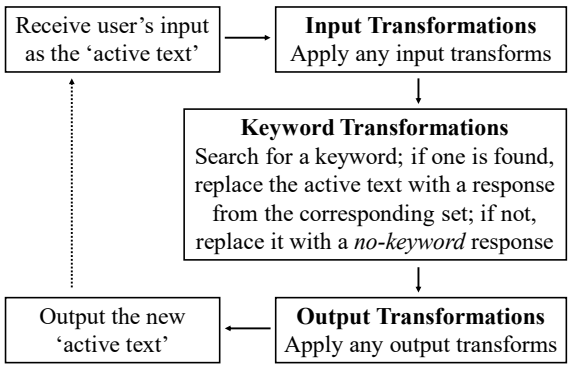
Simple Message Types

- The 'Welcome/Quit' tab shows *Welcome* and *Quitting* messages – one of each is selected respectively to start the conversation, and to end it (when the user selects 'Quit' from the File menu).
- The 'Void/No-key' tab shows *Void Input* messages – one of these is selected in response to any 'null' input – and *No-Keyword* messages – for use when no 'keyword' is identified in the input.
- If there are more than one of any of these kinds of messages, then by default (changeable through the Options menu) the selection is random, except that the same message won't be chosen twice in succession.

9

10

The Main Processing Cycle



10

11

Input/Output Transformations

- Input transformations are applied to the initial input; their main use is to standardise words that you want to be treated similarly, e.g.

```
I mum => mother
```

  if you want 'mum' to be changed to 'mother'.
- Output transformations are applied to the final output; often their main use is to change first-person to second-person and vice-versa, e.g.

```
O i am => YOU ARE
```

- Make sure you capitalise these as illustrated above (this will be explained a bit later).

11

12

Keyword Transformations

- Keywords and responses are grouped into sets, so order them in your script file accordingly (set 1 keys, then set 1 responses, then set 2 keys … etc). Generally it's best to capitalise keys *and* responses.
- Unlike Input and Output Transformations, only *one* Keyword Transformation is applied each time.
- Note how pattern matching and substitution are used within the keywords and responses in the Illustrative Script, and their effect as you 'play'.
- See the help sections on 'The Input/Keyword/Output/ Final Transformation Process' (§ 2.3) and 'Pattern Matching' (§ 3.1) if you want more details.

12

Using the *Elizabeth* Conversation Program

13

Simple Keywords and Responses

- The following script commands create a simple keyword/response set with two keywords and three responses:

```
K MOTHER
K FATHER
 R TELL ME MORE ABOUT YOUR FAMILY.
 R DO YOU HAVE ANY BROTHERS OR SISTERS?
 R ARE YOU THE YOUNGEST IN YOUR FAMILY?
```

- When 'mother' or 'father' is found in the active text, one of the responses will be chosen (randomly, but avoiding immediate repetition if possible).

13

14

Keywords with Substitution

- The following script commands create a keyword/ response set which pattern-matches the keyword against the active text and then makes appropriate substitutions in the response:

```
K [p1] IS YOUNGER THAN [p2]
 R SO [p2] IS OLDER THAN [p1]
```

- Any pattern of the form `[p…]` is a *phrase* wildcard, matching any sequence of *words* (which can contain only letters, hyphens or apostrophes). `[p1]` is treated as a separate pattern from `[p2]`.

14

15

Pattern Matching

- Any of these patterns can be used in combination (see the help file section 'Pattern Matching' for the complete list):
  - `[w…]` any single complete *word* (or part-word)
  - `[t…]` any single complete *term* (or part-term) – a term, unlike a word, may contain digits as well as *letters*
  - `[l…]` any single *letter* (i.e. any character that can occur in a *word*, including hyphen/apostophe)
  - `[p…]` a *phrase* – any sequence of complete *words*
  - `[X…]` any text string which contains only complete 'items' (so it cannot contain only half a word or number).
  - `[b…]` like `[X…]`, but will only match text in which all brackets – '(', ')', '<', and '>', correctly pair up.
  - `[;]` any punctuation mark
  - `[]` matches beginning or end of active text

15

16

Empty Patterns

- `[let1]` and `[let2]` each matches one letter, so the following might generate the dialogue: 'My degree is a BSc. IS GETTING A BSC DEGREE HARD?'

```
K DEGREE [X] B[let1][let2]
 R IS GETTING A B[let1][let2] DEGREE HARD?
```

- Suppose you want to do this not only for 'BSc' and 'BCL' etc, but also 'BA'. To do this, allow the second pattern to match nothing by adding '?':

```
K DEGREE [X] B[let1][let2?]
 R IS GETTING A B[let1][let2?] DEGREE HARD?
```

16

17

Matching the Ends of the Text

- The term `[]` is used to match the beginning, or the end, of the active text. This enables you to treat words differently if they are the first, or last, word of the user's input. We'll see a 'first word test' a bit later (with memorisation of 'my' phrases); here's an example of a 'last word test':

```
O you [] => ME
O you => I
```

- These two output transformations will have the effect of changing 'you' into 'ME' if it is the very last word of the active text, but into 'I' otherwise – this makes sense because when 'you' appears at the end it's normally the *object* of the sentence rather than the *subject* (e.g. 'She saw you').

17

18

Capitalisation and Transformations

- We have seen that different types of capitalisation are typically used for the various transformations:

```
I mum => mother
K FATHER
R TELL ME MORE ABOUT YOUR FAMILY.
O i am => YOU ARE
```

- This all fits with the following rule:

  A lower-case pattern can only match with a lower-case text, whereas an upper-case pattern can match with either a lower-case or an upper-case text.

18

Using the *Elizabeth* Conversation Program

19

- Initially, the input text is converted to lower case. Putting all your input transformations in lower case ensures that the text stays lower case at this stage.
- If a keyword is found, the text usually gets replaced with a response which is *already* in the right form for output, *so you don't want to apply output transformations to it*. This is ensured by putting the responses in upper case, and the left-hand side of the output transformations in lower case.
- If an output transformation is applied, e.g. to change 'my' to 'YOUR', then capitalisation on the right-hand side ensures that no further transformations will be applied to text that's already been converted.
- See § 5.1 for more detail, and worked examples.

19

20

Modularising Your Script

- As your script grows, it can be made easier to manage by dividing it into separate files. See the 'Turbo Eliza' example under the Help menu (Basic script commands).
- You will need one 'master' file, which can then 'pull in' sub-files using an *include directive*, e.g.:

```
#INCLUDE My Scripts\output.txt
```

- This enables you to use e.g. the same set of output transformations within several scripts.
- Sub-files can contain further *include directives*, so you can organise your script into sub-sections, etc.

20

21

Dynamic Commands

- Script commands can be applied dynamically, and can be 'triggered' by almost any kind of process (see the help file on 'Dynamic Script Processing' for details and a variety of examples).
- The most important use of this is for memorisation of phrases, which can then be recalled later, e.g.:

```
K MY NAME IS [phrase]
  & {M [phrase]}
 R NICE TO MEET YOU [phrase]!
N WHAT DO YOU LIKE DOING, [M]?
```

21

22

Memorising and Recalling Phrases

- *Note from the previous example:*
  - '`& {…}`' is used to specify an action, in this case one that is triggered by the matching of a keyword and the selection of a corresponding response;
  - '`{M [phrase]}`' memorises whatever text was matched against `[phrase]`;
  - `[M]` can then be used to recall the latest remembered text, within *any* kind of transformation or response;
  - Here a no-keyword response is created, which when invoked will make use of the latest memory (`[M]`).
  - `[M-1]`, `[M-2]` etc. can be used to recall earlier memories (the last but 1, last but 2, etc.).

22

23

Returning to a Previous Topic

- The most common use of memorisation in the original ELIZA program is to deal with the situation where no keyword is found, to give an impression of continuity by returning to a previous topic.
- A good way of recognising likely topics is to look for user input *starting with* 'my', e.g. 'my dog is ill'. Note the use of `[]` to match the beginning of the text:

```
K [] MY [phrase]
  & {M [phrase]}
 R YOUR [phrase]?
N DOES THAT HAVE ANYTHING TO DO WITH THE
FACT THAT YOUR [M]?
```

23

24

Index Codes

- Every transformation, response, memory etc. that *Elizabeth* accepts is assigned an index code. Unless you specify an index code yourself, these are automatically created for you, starting with '001', '002', '003' etc.
- You can see what index codes have been assigned by inspecting the relevant tables.
- Index codes enable you to pick out specific transformations/ responses/memories for dynamic modification, recall etc.
- We'll be using index codes only for memories – enabling us to handle *many* memories, and not just the latest one. (See help on 'Control of Scripts using Command Index Codes' and 'Command Syntax Reference Guide' for other uses.)

24

Using the *Elizabeth* Conversation Program

25

Memorising Pronoun References

- One simple use of index-coded memories is to keep track of what's been referred to by a recent output, so that pronouns ('it', 'she' etc.) can be dealt with appropriately. The following might yield 'I watch football. WHAT DO YOU THINK OF ARSENAL? They're good. I LIKE THEIR STYLE …': here the input transformations replace 'They're' in the last input with 'ARSENAL ARE', enabling an appropriate response to be found.

```
I THEY'RE => THEY ARE
I THEY => [Mthey]
K FOOTBALL
 R WHAT DO YOU THINK OF ARSENAL?
   & {Mthey ARSENAL}
K ARSENAL
 R I LIKE THEIR STYLE, BUT NOT THEIR RESULTS!
```

25

26

Using Multiple Memories

- This script will keep track of some of your favourites, tell you what they are, and then go on repeating them.

```
W WHAT ARE YOUR FAVOURITE GAME, TEAM AND PLAYER?
K GAME [X?] IS [phrase]
  & {Mgame [phrase]}
K TEAM [X?] IS [phrase]
  & {Mteam [phrase]}
K PLAYER [X?] IS [phrase]
  & {Mplayer [phrase]}
 R THANK YOU - SAY "OK" WHEN YOU'VE FINISHED
K OK
 R YOUR FAVOURITE GAME IS [Mgame], TEAM IS [Mteam],
      AND PLAYER IS [Mplayer]
   & {I [word] => OK}
N PLEASE CARRY ON TELLING ME YOUR FAVOURITES
```

26

27

- *Note from the previous example:*
  - '`K GAME [X?] IS [phrase]`' matches any text containing the word 'GAME' and then at some later point 'IS' followed by a phrase (recall that a 'phrase' here just means one or more words in sequence);
  - '`& {Mgame [phrase]}`' then memorises the relevant phrase under the index code 'game';
  - '`R YOUR FAVOURITE GAME IS [Mgame], TEAM IS [Mteam], AND PLAYER IS [Mplayer]`' outputs the three memories, but *this response cannot be used until something has been memorised under each of the three index codes* (you can check this by inputting 'OK');
  - '`& {I [word] => OK}`' creates an input transformation which changes all words to 'OK' – this simply ensures that from then on, *any* input will be treated as though it was just 'OK OK …'.

27

28

Timing of Dynamic Commands (i)

- In the last example, instead of using 'OK' as a prompt, you might try outputting the three memories, as soon as they exist, using a catch-all output transformation …

```
W WHAT ARE YOUR FAVOURITE GAME, TEAM AND PLAYER?
K GAME [X?] IS [phrase]
  & {Mgame [phrase]}
K TEAM [X?] IS [phrase]
  & {Mteam [phrase]}
K PLAYER [X?] IS [phrase]
  & {Mplayer [phrase]}
 R THANK YOU - DO GO ON ...
O [X] => YOUR FAVOURITE GAME IS [Mgame], TEAM IS
      [Mteam], AND PLAYER IS [Mplayer]
N PLEASE CARRY ON TELLING ME YOUR FAVOURITES
```

28

29

Timing of Dynamic Commands (ii)

- You might now expect that as soon as the three memories have been saved, the catch-all output transformation (`[X] => YOUR FAVOURITE …`) will automatically become operative no matter what the active text is, won't it?
- But doing this *won't* work until you type in *another* input … if you look at the trace tab just after you've typed in your three favourites, you should see why.
- The problem is that each new memory isn't saved until *after* the corresponding response processing has all been done. But the action will work immediately if you insert a '!', e.g.:

```
K GAME [X?] IS [phrase]
  & {!Mgame [phrase]}
```

29

30

Using Null Memories to Keep Track

- Recall that responses (etc.) containing memory references like '`[Mthey]`' cannot be used until those references succeed (i.e. until *something* has been memorised under the relevant code).
- However the 'something' saved can be the null string (i.e. nothing!) – so saving a null memory provides a way of 'keeping track', and controlling which responses (etc.) are used and which are not.
- The advantage of using a null memory is that this can be inserted into any response without affecting what gets output (because, after all, it's the *null* string: it contains no characters at all).

30

Using the *Elizabeth* Conversation Program

31

Changing Mood

- The following script fragment makes *Elizabeth* get progressively more angry at the user's swearing (starting off in the 'calm' state, then progressing to 'cross' and 'enough'; note how 'M\' is used to delete all memories, and that more than one command can be put inside the curly brackets.

```
K DAMN
K BLOODY
 R [Mcalm] I'D RATHER YOU DIDN'T SWEAR, PLEASE
   & {M\
      Mcross}
 R [Mcross] LOOK, JUST STOP SWEARING WILL YOU!
   & {M\
      Menough}
 R [Menough] THAT'S IT! I'VE HAD ENOUGH - GO AWAY!
   & {M\
      O [X] => JUST GO AWAY}
Mcalm
```

31

32

Conditional Commands

- Using null memories to keep track of the 'state' of the conversation is the simplest kind of *conditional* processing.
- You can also define conditional commands explicitly, using angle brackets to specify the relevant condition:

  `<[Mcalm]>: R DON'T SWEAR, PLEASE`

  makes this keyword response available for use only if the memory [Mcalm] is defined

  `<[Mtemper]==CALM>: R DON'T SWEAR, PLEASE`

  makes this keyword response available for use only if the memory [Mtemper] has the value 'CALM' (note that `'!='` *instead of* `'=='` *would check inequality rather than equality)*
- For more on this, see the help sections on 'Giving Direction to a Conversation' and 'Defining and Using Conditional Commands'.

32

33

More on Dynamic Commands

- Almost any script command can be used dynamically, and *virtually* all of them act identically in either case (an exception is when you add a keyword that already exists).
- However for dynamic uses, you will need some commands that you are very unlikely to use directly in a script – for example the commands that *delete* transformations or memories etc. (as we've already used above).
- To test what effect a particular command will have when triggered dynamically, you can type it into the input box, and then press **F1** instead of **Enter**.
- For full details of all available commands, see 'Command Syntax Reference Guide' (§ 6.1 and its subsections).

33

34

Examples of Deletion Commands

- *See the 'Command Syntax Reference Guide' (§ 6.1) for full details of deletion commands, and for the treatment of index codes and keyword sets as mentioned on the next slide.*

`V\ DON'T YOU WANT TO TALK?`
- deletes this specific void input response

`N\`
- deletes all no-keyword responses

`I\ dad => father`
- deletes this specific input transformation

`I\ dad`
- deletes the *first* input transformation whose left-hand side is 'dad'

`K\ MOTHER`
- deletes the keyword 'MOTHER'

`K\ or K/\`
- delete all keywords; 'K/\' deletes all the keyword *sets* too.

34

35

Index Codes and Keyword Sets

- One of the above examples deletes the '*first*' input transformation of a particular kind – when you use any such command, the ordering is *alphabetical by index code*.
- Almost any script command can be assigned an index code when it is created, and this will determine the order in which they are applied and searched for, e.g.:

  `I$first one => two`
  - *defines the input transformation 'one => two' with index code '$first' – '$' comes alphabetically before '0', so this transformation will be done even before the transformation coded '001'. See the help section on 'Alphabets' for details of ordering.*
- Keyword/response sets have index codes, and the keywords/responses also have their own index codes. Within keyword and response commands, '@' can be used to refer to the *current* keyword/response set (i.e. usually, the latest to be modified).

35

36

Commands Within Commands

- Dynamic script commands can be 'nested' like this (note how indentation is used to show the structure):

```
I my sister => my sister
  & {K MOTHER
       & {N TELL ME MORE ABOUT YOUR MOTHER}
     R HOW WELL DO YOUR MOTHER AND SISTER GET ON?}
```

- When the phrase 'my sister' is identified in the input, this adds a new keyword 'MOTHER' together with the response 'HOW WELL …'. But the keyword is also defined in such a way that when it is recognised and the response given, this will trigger another action, creating a no-keyword response 'TELL ME MORE …' which might then be invoked later in the conversation.

36

Using the *Elizabeth* Conversation Program

37

Iteration, Cycling, and Recursion

- Having now covered most of *Elizabeth's* specific commands, it is time to look at the higher-level structures that it provides for handling repeated operations and flow of control.
- Three important concepts here are *iteration*, *cycling*, and (the most significant) *recursion*.
- Within *Elizabeth*, *iteration* and *cycling* apply only to input and output transformations (the latter include also 'final' transformations, which are explained later); *recursion* can apply to any kind of transformation, and is usually most powerful and flexible when used with keyword/response pairs.

37

38

The Concept of Iteration

- The *Oxford Dictionary of Computing* has this to say about iteration:

  [Iteration is] *the repetition of a numerical or non-numerical process where the results from one or more stages are used to form the input to the next. Generally the recycling of the process continues until some preset bound is achieved, or the process result is constantly repeated.*

- Within *Elizabeth*, iteration of input/output/final transformations is automatically halted when they cease to have any effect on the active text (i.e. where 'the process result is constantly repeated').

38

39

Enabling Iteration within *Elizabeth*

- To enable iteration of input transformations, open the Control menu dialog box and click on the 'Permit unlimited iteration' radio button under the *'Input Transformation Control'* heading.
- Now create a script containing only the single input transformation:

  – `I TICK => TICK TOCK`

  and type as input: 'TICK'. There will be a delay while this is processed, and the easiest way to see the output is to look at the 'Dialogue' tab.

39

40

Controlling Loops – the Match Limit

- When iteration is enabled for input transformations, this means that every input transformation will be applied to the active text repeatedly until *either*:
  - the transformation stops having any effect (either because it ceases to apply to the active text, or makes no difference);
  - iteration is terminated for some reason.
- Iteration will *always* be terminated eventually, to prevent the system 'hanging'. The ultimate barrier here is a *match algorithm limit*, which keeps count of how many times the matching algorithm has been called, and prevents further iteration after a certain limit (5,000 by default).
- This limiting value is set through the Control menu dialog.

40

41

The Automatic Undo Facility

- An alternative method of controlling iteration is an *automatic undo facility*, which works on the basis:
  - that any transformation which iterates 10 or more times is 'non-terminating';
  - that any non-terminating transformation should be applied once only.
- To enable the automatic undo facility, un-check the 'Permit Recursion, Text Splitting and …' box in the Control menu dialog. This is necessary because (to avoid control flow conflicts) the automatic undo is not available when recursion is enabled.

41

42

The Concept of Cycling

- The *Oxford Dictionary of Computing* provides this definition of the relevant sense of the word 'cycle':

  [A cycle is] *any set of operations that is repeated regularly and in the same sequence. The operations may be subject to variations on each repetition.*

- Within *Elizabeth*, cycling of input/output/final transformations is subject to the same sorts of limits that apply to iteration; so for example cycling is automatically halted when the cycle of operations ceases to have any effect on the active text.

42

Using the *Elizabeth* Conversation Program

43

Iteration and Cycling

- To appreciate the difference between iteration and cycling, first create a script containing only the two input transformations:
  - `I TICK => TOCK`
  - `I TOCK => TICK TACK`

  and type as input: 'TICK'.
- With iteration enabled but *not* cycling, the output will be 'TICK TACK', each transformation having been applied once. Note that the left-hand side of each does *not* match its right-hand side, so neither of them can be repeatedly applied to its own output.

43

44

Enabling Cycling within *Elizabeth*

- To enable cycling of input transformations, open the Control menu dialog box and click on the 'Permit unlimited cycling' radio button under the *'Input Transformation Cycling'* heading.
- If you do this and then again give the input 'TICK' to the transformations:
  - `I TICK => TOCK`
  - `I TOCK => TICK TACK`

  you will find that the output is much longer, like the output you got earlier with unlimited iteration.

44

45

Controlling Cycling

- Exactly as with iteration, there are two different ways of preventing infinite loops arising from cycling:
  - Further cycling will *always* be prevented eventually, once the *match algorithm limit* is exceeded.
  - If recursion is disabled, it is possible to select the *automatic undo facility*, which works on the basis:
    - that any set of transformations which cycles 10 or more times (this number is adjustable) is 'non-terminating';
    - that any non-terminating set of transformations should be applied once only, so further applications should be 'undone'.
  - As in the case of iteration, the automatic undo facility for cycling is accessible through the Control menu dialog.

45

46

Recursion

- Recursion is a hugely important concept in Computing generally, and especially in Artificial Intelligence. The *Oxford Dictionary of Computing* provides defines *recursion* as:

  The process of defining or expressing a function, procedure, language construct, or the solution to a problem in terms of itself, so producing a recursive function, a recursive procedure etc.
- Within *Elizabeth*, the 'problem' is the generation of appropriate output, and a *recursive* solution is one in which the overall solution is created by 'feeding back' partial solutions into the system.

46

47

Recursion in *Elizabeth*

- *Any text in curly braces* on the right-hand side of an input/output/final transformation rule, or in a keyword response, will be *recursed* – fed back into *Elizabeth* as if it was input:

```
K [word1] [phrase1]
 R {[word1]} {[phrase1]}
K 1
 R ONE
K 2
 R TWO
```

Input of:

`1 2 2 1 2`

will yield output of

`ONE TWO TWO ONE TWO`

- Here the first word in each input is separated from the remainder, and both are recursed in turn. Each single word fails to match with the first keyword, and so 'falls through' to be caught by one of the other keywords and replaced.

47

48

Recursion and Final Transformations

- In the previous example, the input is split into individual words, and each word is then processed separately – through the full input/keyword/output sequence – before the results of this individual processing are recombined in the output.
- *Final transformations* are a special kind of output transformation, differing from the standard kind in being applied *after* recombining has taken place.
- To explore recursion further, with more complex examples, see 'Recursion and Text Splitting' (§ 3.2) and 'The Power of Recursion' (§ 3.3).

48

Using the *Elizabeth* Conversation Program

49

APPENDIX 1: Syntactic Analysis

- The ELIZA method of simple pattern-matching and pre-formed responses may sometimes be able to generate the *illusion* of 'intelligent' language processing, and even in some cases (e.g. a computer help system) provide the basis for a useful tool.
- However to get anywhere near genuine NLP (natural language processing), *Elizabeth* needs to do more than pattern-match – it must be responsive to the *structure* of sentences, and react not just according to the literal word strings they contain, but how these words are put together – their *syntax*.

49

50

A Testbed: Simple Transformations

- A good testbed for *Elizabeth's* potential for handling syntactic structure is the attempt to generate simple grammatical *transformations*.
- A transformation is a change in structure which alters the 'surface' form of the sentence (so the words are different, or in a different order), but *without* significantly altering its 'propositional content' (i.e. what 'facts' are in question; what the sentence 'says' about what or whom).
- Transformations played a major and controversial role in the rise of Chomskyan linguistics, but their value as a useful testbed is independent of all that.

50

51

Our Starting Point: Active Declarative Sentences

- We start from straightforward *active declarative* sentences, such as:
  - John chases the cat
  - The white rabbits bit a black dog
  - You like her
- *Declarative* simply means that these sentences purport to state ('declare') facts – they are not questions or commands, for example.
- Here we shall stick to very simple word categories and grammatical constructs.

51

52

Some Types of Transformation (1): Active to Passive

- Most types of transformation are easier to grasp by example than explanation:
- *Active to Passive*
  - 'John chases the cat' *becomes*
    'The cat is chased by John'
  - 'The white rabbits bit a black dog' *becomes*
    'A black dog was bitten by the white rabbits'
  - 'You like her' *becomes*
    'She is liked by you'

52

53

(2): Yes/No Questions

- These transform the sentence into a question with a simple yes/no answer:
  - 'John chases the cat' *becomes*
    'Does John chase the cat?'
  - 'You like her' *becomes*
    'Do you like her?'
- They can also be applied to passive sentences, though here they're a bit more complicated:
  - 'A black dog was bitten by the white rabbits' *becomes*
    'Was a black dog bitten by the white rabbits?'

53

54

(3): 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:
  - 'John chases the cat' *becomes*
    'John chases the cat, doesn't he?'
  - 'The white rabbits bit a black dog' *becomes*
    'The white rabbits bit a black dog, didn't they?'
  - 'You like her' *becomes*
    'You like her, don't you?'
- These provide an excellent test case, because a tag question must agree with the sentence in *number* (singular or plural), *person* (first person, second, third), *gender* (masculine, feminine, neuter), and *tense* (past, present, future).

54

Using the *Elizabeth* Conversation Program

55

Phrase Structure Grammar (1)

- A common method of syntactic analysis is to break down a sentence into hierarchical components using a *phrase structure grammar*. (Note that here we shall be looking at only a tiny and highly simplified fragment of English, so don't take the rules used here to be absolutely correct or complete!)
- All of the basic sentences we shall be examining consist of a *noun phrase* followed by a *verb phrase*. Crudely, the noun phrase specifies the *subject* of the sentence, e.g. 'John', 'the white rabbits', 'you'. The verb phrase specifies what the subject *does* (or did, or will do), e.g. 'chases the cat', 'like her'.

55

56

Phrase Structure Grammar (2)

- The rule that a sentence can be made up of a noun phrase followed by a verb phrase is represented as:
  S → NP VP
- In the examples we've seen, a noun phrase can be made up in three ways: (a) a single *noun* or *pronoun* (e.g. 'John', 'it'); (b) a *determiner* (or 'article') followed by a *noun* (e.g. 'the rabbits', 'a dog'); (c) a *determiner* followed by an *adjective* followed by a *noun* (e.g. 'the white rabbits'). So:
  NP → N
  NP → D N
  NP → D ADJ N

56

57

Phrase Structure Grammar (3)

- Finally, a verb phrase typically consists of a *verb* followed by a noun phrase, e.g. 'chases …', 'bit …', where the '…' is some noun phrase. So we have:
  VP → V NP

  (We assume here that the verb is a transitive verb: one that has an object as well as a subject. Where a verb is intransitive, the verb phrase can consist of just the verb, e.g. 'sleeps', while many verbs can be either transitive or intransitive, e.g. 'eats'.)
- As we shall see, a set of rules like this can provide a powerful technique for analysing a sentence into its structural components, and *Elizabeth* can help here.
- *See the Elizabeth help on 'Implementing Grammatical Rules' for more discussion and examples of these techniques.*

57

58

Phrase Structure Rules in *Elizabeth*

- The phrase structure rules above can be reversed and then translated into *Elizabeth* input transformations suitable for analysing a sentence into its structural constituents:

  NP → D N

```
I (d:[b1]) (n:[b2]) => (np:(D:[b1]) (N:[b2]))
```

  VP → V NP

```
I (v:[b1]) (np:[b2]) => (vp:(V:[b1]) (NP:[b2]))
```

  S → NP VP

```
I (np:[b1]) (vp:[b2]) => (s: (NP:[b1]) (VP:[b2]))
```

- Note here that a '`[b…]`' pattern can match anything at all, *as long as it contains matching brackets.* This ensures that the sentence structure is recorded by the 'nested' brackets, and that the processing respects this structure.

58

59

- Obviously we also need to specify the categories (noun, verb etc) for the various words. We might end up with a set of input transformations like this:

```
I the => (d:THE)
I dog => (n:DOG)
I cat => (n:CAT)
I chases => (v:CHASES)
I (d:[b1]) (n:[b2]) => (np:(D:[b1]) (N:[b2]))
I (v:[b1]) (np:[b2]) => (vp:(V:[b1]) (NP:[b2]))
I (np:[b1]) (vp:[b2]) => (s: (NP:[b1]) (VP:[b2]))
```

- If we then input the sentence:

```
the dog chases the cat
```

  the input transformations will convert this into:

```
(s: (NP:(D:THE)(N:DOG)) (VP:(V:CHASES) (NP:(D:THE)(N:CAT))))
```

59

60

- Having used the input transformations to analyse the sentence into its constituent structure, we can then apply keyword transformations to alter that structure, e.g. from active to passive:

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

- Then output transformations can be used to decompose the sentence structure back into its parts:

```
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:CHASES passive) => IS CHASED BY
O (d:[b1]) => [b1]
O (n:[b1]) => [b1]
```

- If we then input the sentence:

```
the dog chases the cat
```

  the output will have been 'translated' into the passive form:

```
the cat is chased by the dog
```

60

Using the *Elizabeth* Conversation Program

61

APPENDIX 2: Propositional Logic

- A *proposition* is a statement that some determinate state of affairs is (or is not) the case, e.g. 'Grass is not white', 'Politicians are always liars', '1+1=23', or 'Pigs can fly'. Questions and exclamations are *not* propositions, and it is convenient to restrict our attention to statements that avoid any ambiguity.
- *Propositional logic* deals with reasoning whose logic is analysable *entirely* in terms of whole propositions. This does not include reasoning that depends on propositions' internal structure, e.g. 'Socrates is a man. All men are mortal. Therefore Socrates is mortal.' *(This argument requires Predicate Logic.)*

61

62

Binary Propositional Connectives

- A *binary propositional connective* joins two propositions together to make a third (*complex*) proposition.
- Such connectives in English include 'and', 'because', 'but', 'if', 'implies', 'nevertheless', 'only if', 'or', 'suggests that', 'unless'.
- 'Snow is white' and 'the moon is cheese' are *atomic* propositions (i.e. they're not themselves made up of other propositions). Using the connectives, we get:
  - Snow is white *and* the moon is cheese
  - Snow is white *because* the moon is cheese
  - Snow is white *but* the moon is cheese
  - Snow is white *if* the moon is cheese (*etc.)*

62

63

Truth-Functionality

- A connective # is *truth-functional* if knowing the truth or falsity of *P* and *Q* always gives you enough information to know the truth of falsity of (*P* # *Q*).
- 'And', 'or' are usually treated truth-functionally:

		(*P* and *Q*)	(*P* or *Q*)
P true	*Q true*	*true*	*true*
P true	*Q false*	*false*	*true*
P false	*Q true*	*false*	*true*
P false	*Q false*	*false*	*false*

- But real English isn't so simple, e.g. 'He hit me and I swore at him' gives a different impression from 'I swore at him and he hit me'.

63

64

Translating English Connectives

- Testing arguments using propositional logic requires that *all* connectives be interpreted truth-functionally:
  - Translate 'but', 'nevertheless', 'because' as *and*.
  - Translate 'unless' as *or*.
  - Translate 'if *P* then *Q*', '*Q* if *P*', 'only if *Q* then *P*', 'P only if Q' as *implies*, interpreted like this:

		(*P* implies *Q*)
P true	*Q true*	*true*
P true	*Q false*	*false*
P false	*Q true*	*true*
P false	*Q false*	*true*

64

65

Negating a Proposition

- The *negation* (or *contradictory*) of a proposition is that proposition which denies *exactly* what the first asserts (and therefore asserts exactly what the first denies). So if *P* is true, its negation – not(*P*) – must be false, and if *P* is false, not(*P*) must be true, e.g.:

  P: Snow is white

  not(*P*): Snow is not white *(not* 'Snow is green')

  Q: All cows eat grass

  not(*Q*): Some cow (or cows) doesn't eat grass *(not* 'No cows eat grass'*)*
- Thus *not* is a *unary truth-functional connective*.

65

66

Consistency and Inconsistency

- A set of propositions is *consistent* if it is *logically possible* for all of them to be true together, e.g.

  { The moon is inhabited by spiders ,
  Some man can jump over 100 metres ,
  Examinations are tremendous fun }
- A set of propositions is *inconsistent* if it is *not* logically possible for them all to be true together, e.g.

  { No cow eats aubergines ,
  Daisy is a cow ,
  Daisy eats aubergines }

66

Using the *Elizabeth* Conversation Program

67

Arguments and Validity

- An argument, as the man in Monty Python's Argument Sketch says, is 'a connected series of statements intended to establish a proposition'. The 'statements' [propositions] from which it starts are called its *premisses*, and the proposition which it is intended to establish is called its *conclusion*.
- An argument is *valid* if the truth of the premisses guarantees the truth of the conclusion – that is, *if the premisses are true, the conclusion must be true too.*
- This is the same as saying that *an argument is valid if the set consisting of the premisses, together with the negation of the conclusion, is an inconsistent set.*

67

68

The Resolution Rule

- Suppose these premisses are known to be true:

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

  P must of course be either true or false. If *P* is false, then *Q* must be true (from the first premiss), but if *P* is true, then *R* must be true (from the second premiss). So from these two premisses we can conclude:

  Q or *R*
- This is an application of the rule of *resolution*. In general, this rule lets us move from:

  [A] or *P* or *[B]* *[C]* or not(*P*) or *[D]*

  to *[A]* or *[B]* or *[C]* or *[D]*

68

69

Clausal Form

- A *literal* is either an atomic proposition – a simple proposition we can represent as '*P*' – or else the *negation* of an atomic proposition, e.g. 'not(*P*)'.
- Resolution works on *clauses*: formulae consisting of sequences of literals joined by 'or'. We can simplify the representation of such formulae, e.g. by using '<P>' (with angle brackets) to mean 'not(*P*)', and using commas instead of 'or's.
- Treated like this, 'not(*P*) or *Q* or *R*'
  becomes: '<P>, Q, R'

69

70

Testing Validity by Resolution Refutation

- To test an argument for validity using resolution:
  1. Formalise the propositions of the argument as atomic propositions linked by truth-functional connectives;
  2. Take the (formalised) premisses of the argument together with the negation of the conclusion – we aim to test whether this set of propositions is inconsistent (if it is inconsistent, then the argument is *valid*);
  3. Convert all of the propositions in the set into clausal form (we'll see how to do this soon);
  4. Apply the resolution rule again and again until *either* you can resolve *P* against not(*P*) to get 'NIL' (i.e. a contradiction – the argument's valid) *or* you give up.

70

71

Converting Formulae into Clausal Form

- Remove all 'implies' using the equivalence:
  A implies *B* ≡ (not(*A*) or *B*)
- If at any stage a 'double negation' is produced, replace it with an unnegated formula:
  not(not(*A*)) ≡ *A*
- 'Push the nots inwards' using the identities:
  not(*A* or *B*) ≡ (not(*A*) and not(*B*))
  not(*A* and *B*) ≡ (not(*A*) or not(*B*))
- If any 'and' is left *within the scope* of an 'or', remove it using the identity:
  A or (*B* and *C*) ≡ (*A* or *B*) and (*A* or *C*)

71

72

- If any 'and' remains, it should lie between independent clauses which we can list separately, so we finish up with a straightforward list of such clauses, each consisting purely of 'ored' literals.
- The help file section 'Implementing Propositional Logic' provides an argument to illustrate all this:

G implies (*O* and *B*)	not(*G*) or *O*
	not(*G*) or *B*
B implies *T*	not(*B*) or *T*
(*O* and *T*) implies not(*E*)	not(*O*) or not(*T*) or not(*E*)
E	*E*
therefore not(*G*)	*G* (*negated* conclusion)

72