PROLOG

Sentence Processing

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

Lecture 13

08/11/04

Contents

- Tokenizing a sentence
- Using a DCG to generate queries
- Morphological processing
- Implementing ELIZA
- pattern-matching vs. parsing

Recap I/O commands

write/[1,2]	write a term to the current output stream.
nl/[0,1]	write a new line to the current output stream.
tab/[1,2]	write a specified number of white spaces to the current output stream.
put/[1,2]	write a specified ASCII character.
read/[1,2]	read a term from the current input stream.
get/[1,2]	read a **printable** ASCII character from the input stream (i.e. skip over blank spaces).
get0/[1,2]	read an ASCII character from the input stream
see/1	make a specified file the current **input** stream.
seeing/1	determine the current **input** stream.
seen/0	close the current **input** stream and reset it to user.
tell/1	make a specified file the current **output** stream.
telling/1	determine the current **output** stream.
told/0	close the current **output** stream and reset it to user.
name/2	arg 1 (an atom) is made of the ASCII characters listed in arg 2

Making a tokenizer

- You may remember that our DCGs take lists of words as input:
  - sentence([’I’,am,a,sentence],[]).
- This isn‘t a very intuitive way to interface with the DCG.
- Ideally we would like to input sentences as strings and automatically convert them into this form.
  - a mechanism that does this is called a *tokenizer* (a token is an instance of a sign).
- We introduced all the techniques neccessary to do this in the last lecture.
  - `read/1` accepts Prolog terms (e.g. a string) from a user prompt;
  - `get0/1` reads characters from the current input stream and converts them into ASCII code;
  - `name/2` converts a Prolog term into a list of ASCII codes and vice versa.

Tokenizing user input

- First, we need to turn our string into a list of ASCII characters using `name/2`.

```
|?- read(X), name(X,L).
|: 'I am a sentence.'
L=[73,32,97,109,32,97,32,115,101,110,116,101,110,99,1
   01,46],
yes
```

- This can then be used to look for the syntax of the sentence which identifies the word breaks.
- A simple tokenizer might just look for the ASCII code for a space (32).
- More complex tokenizers should extract other syntax (e.g. commas, 44) and list them as seperate words.
  - syntax is important when it comes to writing a more general DCG.

Tokenizing user input (2)

- As we recurse through the list of ASCII codes we accumulate the codes that correspond with the words.
- Everytime we find a space we take the codes we have accumulated and turn them back into words.
  - we could accumulate characters as we recursed into the program but this would reverse their order (think of `reverse/3`).

```
tokenize([H|T],SoFar,Out):-
    H\==32, tokenize(T,[H|Sofar],Out).
```

  - instead we can recurse within words, adding the characters to the head of the clause and reconstructing the words as we backtrack.
- We stop when we find the end of the sentence (e.g full-stop= 46) or we run out of characters.

An example tokenizer

```
go(Out):-
        read(X),              ←read user input
        name(X,L),            ← turn input into list of ASCII codes
        tokenize(L,Out).      ← pass list to tokenizer

tokenize([],[]):-!.                    ← base case: no codes left
tokenize(L,[Word|Out]):-
        L\==[],
        tokenize(L,Rest,WordChs),      ← identify first word
        name(Word,WordChs),            ← turn codes into a Prolog term
        tokenize(Rest,Out).            ← move onto rest of codes

tokenize([],[],[]):- !.                ← end of word: no codes left
tokenize([46|_],[],[]):- !.            ← full-stop = end of word
tokenize([32|T],T,[]):- !.             ← space = end of word

tokenize([H|T],Rest,[H|List]):-        ← if not the end of a word then add
        tokenize(T,Rest,List).           code to output list and recurse.
```

Example tokenisation

```
| ?- go(Out).
|: 'I am a sentence.'.
Out = ['I',am,a,sentence] ?
yes

| ?- go(Out).
|: honest.
Out = [honest] ?
yes

| ?- go(Out).
|: honest_to_god.
Out = [honest_to_god] ?
yes
```

```
| ?- go(Out).
|:'I, can; contain: (syntax)'.
Out= ['I,','can;', 'contain:',
   '(syntax)'] ?
yes
| ?- go(Out).
|: 'but not apostrophes (')'.

Prolog interruption (h for
   help)? a% Execution aborted
```

- It will also accept compound structures but only if quoted as strings.

```
| ?- go(Out).
|: '[h,j,k,l] blue(dolphin)'.
Out = [‘[h,j,k,l]’,
   'blue(dolphin)']?
yes
```

Tokenizing file input

- We can also convert strings read from a file into word lists.
- Instead of processing a list of characters translated from the user prompt we can read each ASCII code direct from the file
  - get0/1 reads characters direct from an input file and converts them into ASCII code
- Every new call to get0/1 will read a new character so we can use it to process the input file sequentially.
- We could also use full-stops (code 46) to seperate out sentences and generate mulitple sentences in one pass of the file.

From tokens to meaning

- Now we have our word lists we can pass them to a DCG.
- I want to be able to ask the query:

```
|: is 8 a member of  [d,9,g,8].
```

- and for it to construct and answer the query:

```
member(a,[d,9,g,8]).
```

- First my program should ask for input

```
get_go(Out):-
     write('Please input your query'), nl,
     write('followed by a full stop.'), nl,
     tokenize(0,Out),
     sentence(Query,Out,[]), trace,
     Query.
```

- Then parse the word list using a DCG (`sentence/3`) and
- Finally, call the resulting `Query`.

From tokens to meaning (2)

- This is the DCG for this question (it could easily be extended to cover other questions).
  – Word list = [`is,8,a,member,of,[d,9,g,8]].`

```
sentence(Query) --> [is], noun_phrase(X,_),
     noun_phrase(Rel,Y), {Query =.. [Rel,X,Y]}.

noun_phrase(N,PP) --> det, noun(N), pp(PP).
noun_phrase(PN,_) --> proper_noun(PN).

pp(NP) --> prep, noun_phrase(NP,_).

prep --> [of].
det --> [a].
noun(member) --> [member].
proper_noun(X) --> [X].
```

`univ/2` operator creates a predicate

- `Query = member(8,[d,9,g,8]).`

PROLOG

Morphology

- Morphology refers to the patterns by which words are constructed from units of meaning.
- Most natural languages show a degree of regularity in their morphology.
- For example, in English most plurals are constructed by adding ‘s’ to the singular noun

  E.g. program → programs

  lecture → lectures
- These regular patterns allow us to write rules for performing morphological processing on words.
- If we represent our words as lists of ASCII codes then all we have to do is append two lists together:

```
|?-  name(‘black’,L),name(‘bird,L2),
     append(L,L2,L3),name(Word,L3).
L  = [98,108,97,99], L2 = [98,105,114,100],
L3 = [98,108,97,99,98,105,114,100],
Word = blackbird;
```

Pluralisation

- To pluralise a word all we have to do is append the suffix ‘s’ to the word.

```
plural(Sing,Plu):-
    name(Sing,SingChs),
    name(s,PluChs),
    append(SingChs,Suffix,PluChs),
    name(Plu,PluChs).
```

```
|?- plural(word,Plu).
 Plu = words;
 yes
```

- As there are many different morphological transformations in English (e.g. –ed, -ment, -ly) it would be useful to have a more general procedure:

```
generate_morph(BaseForm,Suffix,DerivedForm):-
     name(BaseForm,BaseFormChs),
     name(Suffix,SuffChs),
     append(BaseFormChs,SuffChs,DerFormChs),
     name(DerivedForm,DerFormChs).
```

Pluralisation (2)

```
|?- generate_morph(word,s,Plu).
Plu = words
yes
|?- generate_morph(want,ed,Plu).
Plu = wanted
yes
|?- generate_morph(want,ed,Plu).
Plu = wanted
yes
```

- However, in English, there are many exceptions to the rule…

```
|?- generate_morph(knife,s,Plu).
Plu = knifes → knives
yes
|?- generate_morph(create,ed,Plu).
Plu = createed → created
yes
```

Accommodating Exceptions

- The simplest way to accommodate these exceptions is to encode them explicitly in our rule.
- We can do this by replacing `append/3` with a user-defined predicate `morph/3` that functions in the same way but also catches exceptions.

```
morph("fe","s","ves").          ← if word ends with the characters in
morph("e","ed","ed").             arg1 and suffix is arg2 replace arg1
                                  with arg3.

morph([],Suffix,Suffix).        ← if no exceptions just add suffix.

morph([H|T],Suf,[H|Rest]):-     ← normal recursive append clause;
    morph(T,Suf,Rest).            build result in head.
```

- Remember that double-quotes have the same effect as name/2

```
|?- name('word',L).          |?- L = "word".
L = [119,111,114,100];       L = [119,111,114,100];
yes                          yes
```

ELIZA

- We have already seen how we can use our word lists and a modified DCG to extract rudimentary meaning from sentences but this is not the only way to make our program appear as if it understands.
- In 1966, Weizenbaum wrote a very rudimentary program that used *pattern-matching* to process natural-language input and choose appropriate responses.
- He called his 'chat-bot' ELIZA and modelled her conversational style on that of a Rogerian psychotherapist.

User: You are like my Father in some ways.

ELIZA: WHAT RESEMBLANCE DO YOU SEE?

User: You are not very aggressive but I think you don't want me to notice that.

ELIZA: WHAT MAKES YOU THINK I AM NOT AGGRESSIVE?

User: You don't argue with me?

ELIZA: WHY DO YOU THINK THAT I DON'T ARGUE WITH YOU?

PROLOG

ELIZA (2)

- The role of a Rogerian Psychotherapist is as a non-entity, they should exhibit no persona or knowledge of the world. Therefore, it is the perfect job for a computer.
- All ELIZA needed to do was
  - identify certain patterns of words,
  - extract key elements of the input, and
  - use these elements to construct output based on routine sentence structures.
- Pattern matching doesn’t require that we understand the meaning of the sentence or even its structure. Therefore, we don’t need to parse the sentence using a DCG.
- All we need are lists of common sentences with variable ‘holes’ that can be matched to the input lists.

Implementing ELIZA

- Input: “I think I am depressed.”
- Pattern: [‘I’, X, ’I’, am, Y]
- Output: [why,do,you,X,you,are,Y,’?’]
- = “Why do you think you are depressed?”

Knowledge base contains the fact:

```
rule([‘I’, X, ’I’, am, Y], [why,do,you,X,you,are,Y,’?’]).
```

```
|?- go(Sent), rule(Sent,Resp), writelist(Resp).
   |: ‘I think I am depressed’.
   WHY DO YOU THINK YOU ARE DEPRESSED?

   Resp = [why,do,you,think,you,are,depressed,’?’],
   Sent  = [‘I’, think, ’I’, am, depressed]
   yes
```

Implementing ELIZA (2)

- The more rules you write the greater the range of sentences ELIZA can identify.

```
rule([i,hate,X,'.'], [do,you,really,hate,X,?]).
rule([do,you,Y,me,'?'], [why,do,you,ask,if,'I',Y,you,?]).
rule([i,like,X,'.'],[does,anyone,else,in,your,family,like,X,?])
rule([are,you,X,'?'],[what,makes,you,think,'I',am,X,?]).
rule([you,are,X,'.'],[does,it,please,you,to,believe,'I',am,X,?]
```

- You also need a default response for when ELIZA doesn't recognise the sentence.

```
rule(X,[please,go,on,'.']).
```

- The patterns do not have to be complete. You might only need to match the start of a sentence.

```
rule([i,think|Rest], [why,do,you,think|Rest]).
|: Why do you think people do not like me?
WHY DO YOU THINK PEOPLE DO NOT LIKE ME?
```

← Error: lack of agreement

Post processing

- However, not all sentences can just be mirrored and remain correct.
- For example, pronouns must be reversed as otherwise their referent will change:

```
User: "Do you like me?"
ELIZA: "Why do you ask if you like me?"
```

- We can do this be post-processing the output.
- If our rules preserve original pronouns then we can filter the reply, replacing you for I, me for you, mine for yours, etc.

```
replace([],[]).
replace(['you'|T],['I'|T2]):-
    replace(T,T2).
replace(['me'|T],['you'|T2]):-
    replace(T,T2).
```

Expanding ELIZA's vocabulary

- We can also make our patterns into Prolog rules to allow further processing.
- This allows us to either:
  - accept more than one sentence as input for each pattern
  - `rule([Greeting|Rest],[hi|Rest]):-`
    `member(Greeting,[hi,hello,howdy,'g`day']).`
- or generate more than one response to an input pattern.
  - `rule([Greeting|_],Reply):-`
    `member(Greeting,[hi,hello,howdy,'g`day']),`
    `random_sel(Reply,[[hi],[how,are,you,today,?],`
    `['g`day'],[greetings,and,salutations]]).`

    - Where random_sel/2 randomly selects a response from the list of possibilities.

Pattern matching vs. Parsing

- It looks as if pattern matching is easier to implement than writing a DCG that could handle the same sentences, so why would we use a DCG?

Pattern-Matching

- Pattern-matching needs every possible pattern to be explicitly encoded. It is hard to re-use rules
- Variations on these patterns have to be explicitly accommodated.
- Difficult to build logical representations from constituents without explicitly stating them.
- However, for domains with a limited range of user-input, pattern matching can be sufficient and surprisingly convincing.

DCGs

- A DCG identifies a sentence by fitting it to a structure made up of any range of sub-structures.
- This allows it to identify a wide range of sentences from only a few rules.
- To increase the vocabulary of the DCG you only need to add terminals not whole new rules.
- As the DCG imposes a structure on the sentence it can generate a logical representation of the meaning as a by-product.