CS 3304: Comparative Languages

Project 3

Logical Programming

Due: Apr 14th, 2022 at 11:59 PM on Canvas
Max Points: 100
Total Pages: 4

At some point in your schooling, you may have encountered the concept of *diagramming sentences*. A diagram of a sentence is a tree-like drawing representing the sentence's grammatical structure, including parts such as the subject, verb, and object.

Here is an extended BNF grammar for some simple English sentences that we will use for the purposes of diagramming.

```
<sentence>    -->  <subject> <verb_phrase> <object>
<subject>     -->  <noun_phrase>
<verb_phrase> -->  <verb> | <verb> <adv>
<object>      -->  <noun_phrase>
<verb>        -->  defined in the Verbs list loaded via the definitions.pl
<adv>         -->  defined in the Adverbs list loaded via the definitions.pl
<noun_phrase> -->  [<adj_phrase>] <noun> [<prep_phrase>]
<noun>        -->  defined in the Nouns list loaded via the definitions.pl
<adj_phrase>  -->  <adj> | <adj> <adj_phrase>
<adj>         -->  defined in the Adjectives list loaded via the definitions.pl
<prep_phrase> -->  <prep> <noun_phrase>
<prep>        -->  defined in the Prepositions list loaded via the definitions.pl

The parts within the [ ] are optional.
```

This grammar can generate an infinite number of sentences. One sample is:

```
mean cow saw carefully green alice with book
```

(For simplicity, we ignore articles, punctuation, and capitalization, including proper nouns and the first word of the sentence.)

Implementation Language

For this project, we are going to be using Prolog. Your solution may only use standard Prolog code.

This assignment will also give you more experience using BNF. Decide what basic operations you need to implement, how those will be written as predicates, and how they can be used to build higher-level predicates.

Input Format

Your program should read "candidate sentences", one per line, from a file named "input.txt". For each input line, your program should attempt to interpret the line's contents (a whitespace-separated list of words) as a sentence. You will need to read each entire line as a string. This string will have to be split into tokens, then feed them to your parsing routine. See the provided starter code for how to do this.

If there is another sentence after the current sentence, it will be followed by a space and then a period. If there are no sentences after the current one, the sentence will end with a new line.

Output Format

The output of your program should be produced in a file named "output.txt". Your program should produce a "diagrammed" version of the input string, which means a sentence is in properly delimited form. "Properly delimited" means that each non-terminal appearing in the input string now has a set of delimiters around it For instance, the input string:

```
"alice found mean green book"
```

would be delimited as

```
( [ { alice } ] ) [ % found % ] ( [ < mean > < green > { book } ] )
```

A more complicated example is

```
"mean cow saw carefully green alice with book"
```

which returns

```
( [ < mean > { cow } ] ) [ % saw % $ carefully $ ] ( [ < green > { alice } * with * { book } ] )
```

In addition, there are two distinct error conditions that your program must recognize. First, if a given string does not consist of valid tokens, then respond with this message:

```
"input has invalid tokens"
```

Second, if the parameter is a string consisting of valid tokens but it is not a legitimate sentence according to the grammar, then respond with the message:

```
"input is not a sentence"
```

Note that the "invalid tokens" message **takes priority** over the second message; the "not a sentence" message can only be issued if the input string consists entirely of valid tokens.

Details

Let me see if I can shed some light on the various delimiters in the output above. I have tried to use some logic on how they are applied and help indicate the parts. So the subject and object each directly contain a noun phrase, so the ( ) on those parts indicate the subject or object phrase. Then the noun phrase and verb phrases are contained within [ ]. From there, each of the individual parts is contained within unique delimiters to set them apart from the other parts:

Part	Delimiters
subject	( )
object	( )
noun phrase	[ ]
verb phrase	[ ]
adjective phrase	none
prep phrase	none
noun	{ }
verb	% %
adjective	< >
adverb	$ $
preposition	* *

Running/Testing your program

While you are working on your predicates, I recommend using `swipl` to load your file and make sure the smaller parts work. Once you are ready to test the entire program, from the terminal you can run this command:

```
swipl -f starter.pl
```

and it will run `swipl` with the file named `starter.pl`, use your filename if you didn't call it `starter.pl` then write `make_go_now()` in the shell to meet that goal and if it is successful it will return true.

This is the command I will be using to test your code. So make sure you have the `make_go_now` predicate that makes all of your code go.

Submitting Your Program

You will submit your code, `starter.pl` file, to Canvas for grading.

Files Attached:

- `starter.pl`
- `definitions.pl`
- `input.txt`
- `output.txt`
- `input_2.txt`
- `output_2.txt`
- `input_3.txt`
- `output_3.txt`

< - - - - - - - - - End of the document - - - - - - - - - >