CCM Programming Challenge: Dotsville Implementation, Part 4

This programming challenge pertains to the implementation of an **interpreter** for for Dotspeak, a little language that interfaces with Dotsville.

Please ...

1. Be sure that you have completed Part 3 of the Dotsville implementation exercise prior to commencing this part, Part 4, of the Dotsville implementation exercise.
2. Please work by analogy with the Purple interpreter presented in class, being mindfull of the fact that:
   (a) You will be judiciously adding **semantic augmentations** to the DCG that was written for the recognizer.
   (b) You will be referencing some **low-level semantic predicates** directly in your "escape from DCG" grammar augmentations, predicates from `dotsville.pro` (`display_world` and `clear_world`) and from `dotsville_lls.pro` (e.g., `sprinkle_one_dot`).
   (c) You will be writing **mid-level semantic predicates** within the file featured in this assignment (e.g., `is_there_a_dot` and `list_the_dots` and `add_one_colored_dot(Color)`) that will call low-level semantic predicates from your other assignments (e.g., `add_dot(Cell,Color)`) and `exists_dot_of_color_on_table` and `list_dots_in_column(Column)` and `available_spaces(S)`).
3. Some bits of this assignment will be quite straightforward. Others, not so much. You are encouraged to adopt a process of doing easier things first, and verify that things are done correctly, just as Descartes suggested one should do in his Discourse on Method!

Tasks

Tasks 1 through 20 are very specific! Task 21 is very general! This approach is designed to afford everyone to have at least some success with this assignment. The remaining three tasks are fairly clerical in nature.

It is expected that you work out issues on your own, to the degree that you can, once you get beyond the tasks that are subsumed by the ONE BIG HINT which I am going to provide you.

The **one big hint** will carry you through the first 10 tasks, and should give you a good understanding of how to proceed with this assignment. You should consider each of the second 10 tasks as a puzzle to be solved. These are relatively straightforward puzzles. Beyond that, when you are asked to continue selecting and implementing forms on your own, do your best to order the "puzzles" – the implementation tasks – from easy to hard, and do as many of them as you are capable of doing.

1. Copy `dotsville_r.pro` to a file called `dotsville_i.rp` in the same directory, and then:
   (a) Change the comments appropriately.
   (b) Add an import for `dotsville_lls.pro`.
   (c) Change the `recognizer` predicate to the corresponding `interpreter` predicate.

   Run the `interpreter` predicate, and make sure it works just as the `recognizer` predicate worked in Part 3.
2. Add a sentence to the Dotspeak language so that you can display the world. And, test to see that it works!
3. Implement the semantics for the "display the world" command. Test it!
4. Implement the semantics for the "stop" command. Test it!

5. Implement the semantics for the "fill the world" command. Test it!
6. Implement the semantics for the "clear the world" command. Test it!
7. Implement the semantics for the "add nsdot" form. Test it!
8. Implement the semantics for the "is there nsdot" form. Test it!
9. Implement the semantics for the "list the dots" command. Test it!
10. Implement the semantics for the "add csdot" form. Test it!

11. Implement the semantics for the "is there nsdot on the table" form. Test it!
12. Implement the semantics for the "is there nsdot not on the table" form. Test it!
13. Implement the semantics for the "is there csdot" form. Test it!
14. Implement the semantics for the "is there csdot on the table" form. Test it!
15. Implement the semantics for the "add csdot to column columnid" form. Test it!
16. Implement the semantics for the "how many dots in column columnid" form. Test it!
17. Implement the semantics for the "how many dots in the world" form. Test it!
18. Implement the semantics for the "is column columnid empty" form. Test it!
19. Implement the semantics for the "is column columnid full" form. Test it!
20. Implement the semantics for the "remove a dot from column columnid" form. Test it!

21. **Continue to implement the remaining forms! As many as you can!! Repeatedly, select a form to implement, and then test your implementation!**

22. Prepare a document which lists all of the forms (sentential forms and sentences) that you sucecssfully implemented, including the first 10, and the second 10, and all of the rest that you managed to implement.
23. Prepare a demo which features at least one sentence corresponding to each form that you implemented, in an order that is consistent in the order that you implemented them.
24. Post your source code, the document listing the forms that you implement, and the domo to your web work site, **by class time on Friday, November 20, 2020**.

One Big Hint

I am going to share my work for the first 10 tasks. More significantly, I think, I am going to share some of my thinking, (in green), for the tasks. Perhaps you will learn something useful from this "learn to think by example" mode of instruction.

1. **Transition from the recognizer to the interpreter.** Copy `dotsville_r.pro` to a file called `dotsville_i.rp`, change the comment, add import for `dotsville_lls.pro`, change the `recognizer` predicate with the corresponding `interpreter` predicate. Then, run the `interpreter` predicate to make sure it works just as the `recognizer` predicate worked in Part 3.

   When you can decompose a problem into separable parts, it is a good idea to keep them separate. I will let the recognizer (`dotsville_r.pro`) remain as it is as an artifact, and commence further work in a variant of the that contains essentially the same code (`dotsville_i.pro`) but which is altered slightly with future work in mind. Of course, I will want to test the new file to be sure I did not damage in moving from the one file to the other file.

THE TEST ...

```
bash-3.2$ swipl
...

?- consult('dotsville_i.pro').
%  ../libraries/io/io.pro compiled 0.00 sec, 29 clauses
%   dotsville.pro compiled 0.00 sec, 34 clauses
%   ../libraries/lp/lp2020.pro compiled 0.00 sec, 67 clauses
%   ../libraries/gv/gv.pro compiled 0.00 sec, 12 clauses
%  dotsville_lls.pro compiled 0.01 sec, 205 clauses
% dotsville_i.pro compiled 0.01 sec, 357 clauses
true.

?- interpreter.
|: is there a blue dot?
|: display the world.
Error ...
|: display world.
Error ...
|: list the dots.
|: list the blue dots.
|: stop.
|: go.
Error ...
|:   C-c C-c
Action (h for help) ? a
abort
% Execution Aborted
?-
```

2. **Add a DCG rule to recognize "display the world".**

   Where might be the best spot in the DCG to place this Dotspeak sentence? The obvious place is amid the `displaycommand` rules! Since the command is so useful, I will place to be the first of the variants.

```
displaycommand --> [display], [the], [world].
displaycommand --> [list], [the], [dots].
displaycommand --> [list], [the], color, [dots].
...
```

THE TEST ...

```
bash-3.2$ swipl
...

?- interpreter.
|: display the world.
|: display world.
Error ...
|:   C-c C-c
Action (h for help) ? a
abort
```

```
% Execution Aborted
?-
```

3. **Implement the "display the world" command.**

   This will be easy, since there is a predicate appearing in `dotsville.pro` to do just this task! I don't need any information beyond the syntax of the form, so I can simply add an "escape from Prolog" augmentation to the relevant grammar rule, and call the predicate to display the world from within it.

```
displaycommand --> [display], [the], [world],
   {display_world}.
```

   THE TEST ...

```
bash-3.2$ swipl
...

?- interpreter.
|: display the world.




- - - - - -
|:   C-c C-c
Action (h for help) ? a
abort
% Execution Aborted
?-
```

4. **Implement the "stop" command.**

   This will be really easy! I can do what was done in the Purple interpreter!

```
displaycommand --> [stop],
   {abort}.
```

   THE TEST ...

```
bash-3.2$ swipl
...

?- interpreter
|: stop.
% Execution Aborted
?-
```

5. **Implement the "fill the world" command.**

   This will be just as easy as "display the world" was, since there is a predicate appearing in `dotsville_lls.pro` to do just this task! As in that case, I don't need any information beyond the syntax of the form, so I can

simply add an "escape from Prolog" augmentation to the relevant grammar rule, and call the predicate to fill the world from within it.

```
displaycommand --> [display], [the], [world],
   {fill_world}.
```

THE TEST ...

```
bash-3.2$ swipl
...

?- interpreter.
|: fill the world.
|: display the world.
P B Y G B B
G Y R O P O
G R G P R B
Y Y R P R G
- - - - - -
|: stop.
% Execution Aborted
?-
```

6. **Implement the "clear the world" command.**

   This is a simple riff on "display the world". In fact, the useful predicate for this tasks appears in the same file as it did for that task, that is, in `dotsville.pro`.

```
displaycommand --> [clear], [the], [world],
   {clear_world}.
```

THE TEST ...

```
bash-3.2$ swipl
...

?- interpreter.
|: fill the world.
|: display the world.
R P O G Y G
O G P R B Y
Y Y O G B Y
R O P O R R
- - - - - -
|: clear the world.
|: display the world.



- - - - - -
|: stop.
```

```
% Execution Aborted
?-
```

7. **Implement the "add nsdot" form.**

   It turns out that this one is east, too. But only because I recall one particular predicate lurking in the `dotsville_lls.pro` code! If I didn't remember that little program, I would have to do work. **It might be a good idea to write down a list of all of the potentially useful predicates in the `dotsville.pro` file and the `dotsville_lls.pro` file – so that I won't fail to avail myself of them when the might come in handy!**

   There is no information needed to implement the semantics, apart from the syntax of the sentence itself, so a simple "break from DCG" annotation will do the job, with `sprinkle_one_dot` from `dotsville_lls.pro` being the featured predicate in the annotation.

```
displaycommand --> [clear], nsdot,
   {sprinkle_one_dot}.
```

THE TEST ...

```
bash-3.2$ swipl
...

?- interpreter.
|: add a dot.
|: add a dot.
|: add a dot.
|: display the world.


P
Y B
- - - - - -
|: stop.
% Execution Aborted
?-
```

8. **Implement the "is there nsdot" form.**

   There is no information needed from the recognition process, which makes the task very simple in one respect. We will once again be able to simply add a "break from DCG" augmentation to the grammar rule.

   However, there is not such an predicate lying around in the code for either of the first two parts of this assignment to do just what is needed, which is to say "Yes" or "No" depending on whether there is a dot in the world. Consequently, we will have to write a **midlevel semantic predicate** to do the job!

   It occurs to me that if there is, indeed, a dot in the world, it will be lying on the table, due to the property of the world that dots cannot float in space. And I also recall that there is a predicate from `dotsville_lls.pro` that pertains to checking to see if there is a dot on the table. It may not be "perfect" for the job, but it will do!

   I will choose a nice name for my midlevel predicate, make use of it in the DCG augmentation, and then define it near the end of the file in an appropriately labelled section of the code.

```
existentialquestion --> [is], [there], nsdot,
   {is_there_a_dot}.



...

% ------------------------------------------------------------------------
% Additional semantic support
% ------------------------------------------------------------------------

% ------------------------------------------------------------------------
% is_there_a_dot

is_there_a_dot :-
  exists_dot_of_color_on_table(_),
  write('Yes.'),nl.
is_there_a_dot :-
  write('No.'),nl.
```

THE TEST ...

```
bash-3.2$ swipl
...

?- interpreter.
|: display the world.




- - - - - -
|: is there a dot?
No.
|: add a dot.
|: add a dot.
|: display the world.


    G
    R
- - - - - -
|: is there a dot?
Yes.
|: stop.
% Execution Aborted
?-
```

9. **Implement the "list the dots" command.**

   I can do this one in roughly the same way that I did the previous one, since the conditions are the same (no data needed as a result of the recognition, yet no perfect low-level semantic support).

I will choose a nice name for a midlevel semantic predicate, make use of it in the DCG augmentation, and then define the midlevel semantic predicate in the "semantic support" section of this file, making judicious use of functionality from the `dotsville_lls.pro` code to make the task relatively simple. In particular, I will use `list_dots_in_column(Column)` to list the dots column by column, from column 1 to column 6.

```
displaycommand --> [list], [the], [dots],
   {list_the_dots}.



...

% ------------------------------------------------------------------------
% Additional semantic support
% ------------------------------------------------------------------------

...

% ------------------------------------------------------------------------
% list_the_dots

list_the_dots :-
  list_dots_in_column(1),
  list_dots_in_column(2),
  list_dots_in_column(3),
  list_dots_in_column(4),
  list_dots_in_column(5),
  list_dots_in_column(6).
```

THE TEST ...

```
bash-3.2$ swipl
...

?- interpreter.
|: add a dot.
|: add a dot.
|: add a dot.
|: add a dot.
|: add a dot.
|: display the world.

      B
B     G
G     O
- - - - - -
|: list the dots.
dot(cell(1,1),color(green))
dot(cell(2,1),color(blue))
dot(cell(1,4),color(orange))
dot(cell(2,4),color(green))
dot(cell(3,4),color(blue))
```

```
|: stop.
% Execution Aborted
?-
```

10. **Implement the "add csdot" form.**

    This task is quite a bit more interesting than any of the previous tasks, since data (one bit of data, anyway) must be gathered during the recognition process. To be precise, for this form, the color of the "color specific dot" must be determined and made accessbble for subsequent use. In order to gather this data, the "extra parameter" augmentation of DCG comes into play.

    Once the process of recognition succeeds and gathers the needed bit of data, that data is used as a parameter to the midlevel semantic predicate (`add_one_colored_dot(Color)`) defined for the need at hand! This midlevel semantic predicate is a simple riff on the `sprinkle_one_dot` predicate from the `dotsville_lls.pro` file.

```
addcommand --> [add], csdot(Color),
   {add_one_colored_dot(Color)}.

csdot(Color)--> detcolor(Color), [dot].

detcolor(Color)--> [a], nonorange(Color).
detcolor(orange)--> [an], [orange].

nonorange(red)--> [red].
nonorange(blue)--> [blue].
nonorange(yellow)--> [yellow].
nonorange(purple)--> [purple].
nonorange(green)--> [green].



...

% ------------------------------------------------------------------------
% Additional semantic support
% ------------------------------------------------------------------------

...

% add_one_colored_dot(Color)

add_one_colored_dot(Color) :-
  available_spaces(S),
  pick(S,space(Row,Column)),
  add_dot(cell(Row,Column),color(Color)).
add_one_colored_dot(_) :-
  write('### Could not add the dot.'),nl.
```

THE TEST ...

```
bash-3.2$ swipl
```

```
...

?- interpreter.
|: display the world.




- - - - - -
|: add a red dot.
|: display the world.



    R
- - - - - -
|: add a blue dot.
|: display the world.



    R   B
- - - - - -
|: add a yellow dot.
|: display the world.



    R   B Y
- - - - - -
|: add a green dot.
|: add a purple dot.
|: add an orange dot.
|: display the world.


    G   P O
    R   B Y
- - - - - -
|: stop.
% Execution Aborted
?-
```