HW 2

Posted Friday, September 19

Due Thursday, October 2 at midnight

Submit files `part1.pl` and `part2.pl` in Submitty. There are 25 autograded points for Part 1, 15 for Part 2, and 10 TA points for code clarity and comments.

Part 1: Answering Questions about Board Games

There's a board game café in Troy, NY which has over 1000 games in their inventory. This makes it overwhelming when trying to decide which game to play. Luckily, there are real life constraints to help narrow this down. For part 1 of this homework, you will be implementing a Horn clause parser which takes a natural language query and produces a response if the query is well-formed. Your parser must handle exactly these four query patterns:

1. **"which games take less than X minutes"** (plural)
2. **"which game takes less than X minutes"** (singular)
3. **"which games allow for X players"** (plural)
4. **"which game allows for X players"** (singular)

Knowledge Base

We compiled a knowledge base in Prolog `https://www.cs.rpi.edu/~milanova/csci4430/games_small.pl` which includes facts about our games. They will be of the following format:

```
game(quoridor).
min_players(quoridor, 1).
max_players(quoridor, 4).
playing_time(quoridor, 15).
```

Expected Behavior

For a given query, the answer should be formatted as the following if accepted by the grammar:

- If no games match: `none`
- If one game matches, return the single game: `azul`
- If more than one matches and the question was plural, then return a comma-separated list where the games are sorted in alphabetical order: `azul, splendor`
- If more than one matches and the question was singular, then return the game that's first in the sorted order: `azul`

Grammar Specification

The grammar below specifies the four types of natural language queries.

$$\mathit{question}(\text{Query}, \text{Plurality}) \rightarrow \texttt{which}\ \mathit{noun\_phrase}(\text{Plurality})\ \mathit{verb\_phrase}(\text{Plurality}, \text{Query})$$

$$\mathit{noun\_phrase}(\text{singular}) \rightarrow \texttt{game}$$
$$\mathit{noun\_phrase}(\text{plural}) \rightarrow \texttt{games}$$

$$\mathit{verb\_phrase}(\text{singular}, \text{player\_query}(\text{N})) \rightarrow \texttt{allows for}\ \mathit{number}(\text{N})\ \texttt{players}$$
$$\mathit{verb\_phrase}(\text{plural}, \text{player\_query}(\text{N})) \rightarrow \texttt{allow for}\ \mathit{number}(\text{N})\ \texttt{players}$$

$$\mathit{verb\_phrase}(\text{singular}, \text{time\_query}(\text{N})) \rightarrow \texttt{takes}\ \mathit{time\_comparison}(\text{N})$$
$$\mathit{verb\_phrase}(\text{plural}, \text{time\_query}(\text{N})) \rightarrow \texttt{take}\ \mathit{time\_comparison}(\text{N})$$

$$\mathit{time\_comparison}(\text{N}) \rightarrow \texttt{less than}\ \mathit{number}(\text{N})\ \texttt{minutes}$$

Implementation

Step 1: Implementing the Grammar in Prolog

Each rule in the grammar must be implemented in Prolog using *difference lists*. Do **not** use the built-in Definite Clause Grammar parser (DCG is essentially syntactic sugar over difference lists with some additional functionality). For example, the top-level rule shoult be written as

```
question(Query, Plurality, S0, S) :-
    word(which, S0, S1),
    noun_phrase(Plurality, S1, S2),
    verb_phrase(Plurality, Query, S2, S).
```

Keep in mind that there are two possible assignments to `Plurality`, `singular` and `plural` as specified in the grammar. Also, for `Query` there are two possibilities `time_query(N)` and `player_query(N)`. Predicate `question` is called with a bound input `S0` and it populates `Query` and `Plurality` if a parse exists. `S` is the rest-of-list. An example question parse is

```
?- question(Query, Plurality, [which, game, takes, less, than, '10',
   minutes], []).
Query = time_check(10),
Plurality = singular ;
false.

?- question(Query, Plurality, [which, game, take, less, than, '10',
   minutes], []).
false.
```

You may find predicate `atom_number` useful; it converts an atom into its numerical value.

Step 2: Answering the Query

Write the implementation of `answer_query(Query, Answer)` which takes the parsed query, and binds `Answer` to games:

```
?-
?- answer_query(time_query(20),Answer).
```

```
Answer = cinco_linko ;
Answer = animal_yahtzee ;
Answer = big_boggle ;
Answer = quoridor ;
false.
```

Step 3: Main Question–Answer Interface

Write the predicate `ask_question(QuestionString, Answer)` which takes a string representing the question, converts to lowercase, tokenizes it, runs the parser, and then answers the query. Note that now the answer must be formatted as described in the expected behavior section. Recall that the response needs to be in **alphabetical order**.

Example usage:

```
?- ask_question("which game allows for 5 players", Answer).
Answer = big_boggle .

?- ask_question("which games take less than 20 minutes", Answer).
Answer = 'animal_yahtzee, big_boggle, cinco_linko, quoridor' .

?- ask_question("which games take less than 5 minutes", Answer).
Answer = none .

?- ask_question("which games allows for 5 players", Answer).
false.
```

Some useful predicates are `downcase_atom` (turns a string into lower-case), `atomic_list_concat` (tokenizes a string and vice versa), "imperative" `findall` (bags all query answers), and `sort`.

Part 2: Circular Sitting

This problem is to find all sitting arrangements on a circular table. A sitting arrangement is represented with a Prolog list, e.g., `[a,b,c,d]` is an arrangement of four guests. Let the first one be an "anchor" that does not move. We are allowed to construct a new arrangement by the transformation that swaps two guests next to each other (other than the anchor). E.g., we can transform `[a,b,c,d]` into `[a,c,b,d]` and into `[a,b,d,c]`. It turns out that applying this transformation repeatedly will find all possible circular sitting arrangements.

Step 1: Transform

Write predicate `transform(A,N)` that takes a bound list `A` of arrangements and generates new ones by applying all possible swaps:

```
?- transform([a,b,c,d],N).
N = [a, c, b, d] ;
N = [a, b, d, c] ;
false.
```

Step 2: Generation as Reachability

We will model the generation of arrangements as a reachability problem. Write a predicate `path(X,Y)` that evaluates to true if either 1) `X = Y`, i.e., there is a path from `X` to itself, or 2) there is a `Z` such that there is a path from `X` to `Z` and `Z` can be `transform`-ed by a swap into `Y`. Next, write a one argument predicate `write_all` that uses `path` to list all arrangements with no repetition. Note that for this assignment it is **not** acceptable to precompute the arrangements some other way, you do need to generate them with `path`. You may want to use predicates `write` and `nl`. The following is what the autograder expects

```
?- write_all([a,b,c,d]).
[a,b,c,d]
[a,c,b,d]
[a,c,d,b]
[a,d,c,b]
[a,d,b,c]
[a,b,d,c]
true .
```