

CSI 3120

Lab 8 Report

Group #6

Carl Li 300235679

Ruoyu Liu 300176134

Roy Rui 300176548

TASK A: Travel Planner

1. Dynamic Predicates:

```
:- dynamic destination/4.  
:- dynamic expense/3.
```

This allows the use of `assert` and `retract` to dynamically add or remove facts during runtime, ensuring that destinations and expenses can be updated as needed.

```
add_destination(Name, StartDate, EndDate, Budget) :-  
    assert(destination(Name, StartDate, EndDate, Budget)).  
  
remove_destination(Name) :-  
    retract(destination(Name, _, _, _)),  
    retractall(expense(Name, _, _)).
```

The `assert` predicate adds a new destination fact dynamically, including details like name, start and end dates, and budget.

The `retract` predicate removes the destination and associated expenses, ensuring consistency in data.

2. Recursion and Backtracking:

The recursion is used in calculating expenses for a destination, allowing traversal through all relevant expense facts.

The code uses the `cut (!)` operator to avoid unnecessary backtracking. But this time we use `findall/3` to avoid backtracking.

3. DCG Implementation:

The example code is as shown:

```
command(add_dest(Name, Start, End, Budget)) -->
    ["add"], ["destination"], [Name], [Start], [End], [Budget].

command(remove_dest(Name)) -->
    ["remove"], ["destination"], [Name].

command(list_expenses(Dest)) -->
    ["list"], ["expenses"], [Dest].

command(check_budget(Dest)) -->
    ["check"], ["budget"], [Dest].
```

4. File I/O and Error Handling:

save_journey(Filename): This predicate is used to save the current destination and expense information to the specified file. open/3 and close/1 are used to open and close the file, and write_destinations/1 and write_expenses/1 are responsible for writing the data to the file.

write_destinations(Stream) and write_expenses(Stream): Use format/3 to format each piece of information about the destination and expense and write it to the file.

Error handling:

exists_file/1 is used to check if the file exists, and see/1 and seen/0 are used to open and close the file for reading.

clear_data/0 is used to clear all currently stored destinations and expenses before loading new data.

5. Testing and Results:

```
% c:/Users/13570/Desktop/lab8.pl compiled 0.00 sec, 22 clauses
?- add_destination('Paris', '2024-11-01', '2024-11-10', 2000).
true.
```

Add expense:

```
?- add_expense('Paris', 'Food', 300).
true.

?- add_expense('Paris', 'Living', 500).
true.

?- add_expense('Paris', 'Trans', 200).
true.
```

Budget valid: pass

```

-----
?- validate_budget('Paris').
true.

```

Budget valid: fail

```

?- add_destination('Ottawa', '2024-09-01', '2024-09-05', 500).
true.

?- add_expense('Ottawa', 'Trans', 200).
true.

?- add_expense('Ottawa', 'Food', 400).
true.

?- validate_budget('Ottawa').
false.

```

Filtering Destinations and Expenses

```

?- add_destination('Paris', '2024-11-01', '2024-11-10', 2000).
true.

?- add_destination('London', '2024-12-01', '2024-12-15', 1500).
true.

?- add_destination('Berlin', '2024-10-01', '2024-10-15', 1800).
true.

?- filter_destinations_by_date('2024-10-01', Destinations).
Destinations = ['Berlin'].

?- filter_destinations_by_date('2024-11-01', Destinations).
Destinations = ['Paris', 'Paris'].

?- filter_destinations_by_date('2024-12-01', Destinations).
Destinations = ['London'].

?- add_expense('Rome', 'Hotel', 400).
true.

?- add_expense('Rome', 'Food', 300).
true.

?- add_expense('Rome', 'Transport', 150).
true.

?- filter_expenses_by_category('Food', Expenses).
Expenses = [('Paris', 300), ('Ottawa', 400), ('Rome', 300)].
~ ■

```

Command Parsing with DCGs:

```

?- parse_command("add destination Paris 2024-11-01 2024-11-10 2000", Command).
Command = add_dest("Paris", "2024-11-01", "2024-11-10", "2000").
?- ■

```

Saving and Loading Journey (File I/O)

```

?- save_journey('test_journey.txt').
true.

?- load_journey('test_journey.txt').
true.

?- destination('Paris', StartDate, EndDate, Budget).
StartDate = 2024-11-1,
EndDate = 2024-11-10,
Budget = 2000 ,

?- expense('Paris', 'Transport', Amount).
Amount = 300 ,

?-

```

Reference:

1.The use of assert and retract in prolog, and how I use these two to deal with ideas

ChatGPT: In Prolog, assert and retract are built-in predicates for dynamically managing facts and rules. They allow you to modify the database while the program is running, which is very useful when managing data that needs to be added, deleted, and updated frequently.

assert(destination(Name, StartDate, EndDate, Budget)): When you want to add a new destination, you call add_destination/4. This predicate uses assert to store the destination information (name, start date, end date, and budget) into the database. This way, the new destination can be referenced and used in subsequent operations of the system.

2. How to achieve the goal to filter destination by date and expense?

ChatGPT: Your goal is to implement filtering for destinations and expenses, filtering by date range or expense category, respectively. Specifically, you need to implement the following two predicates:

Filter destinations by date Filter expenses by expense category Filter destinations by date to find all destinations within a specified time period.

Filter expenses by category to find all expense information belonging to a specific category. In Prolog, we usually use predicates such as findall to collect all matching results that meet a certain condition and store them in a list. In filter_destinations_by_date/2, findall(Name, Condition, Destinations) is used to collect all Names that meet the condition and then store these names in the Destinations list.

In filter_expenses_by_category/2, findall((Dest, Amount), expense(Dest, Category, Amount), Expenses) is used to find all expenses that match the specified Category and store them in the Expenses list. You implemented filtering in filter_destinations_by_date/2, matching the Start and End date ranges to ensure that only destinations that meet the condition are added to the results.

@>= and @<= are standard comparison operators used to compare the order of dates, so that you can check whether a given Date is between the start and end dates.

In this part of the code, one benefit of using `**findall/3**` is that it avoids unnecessary backtracking. In Prolog, backtracking is usually used to find all possible solutions, and `findall` collects all qualifying results at once, so there is no need to backtrack, which can significantly simplify the code. The title requirement mentions "Use the cut operator to control backtracking." In this case, since `findall/3` has effectively controlled backtracking, no additional cut is required in your implemented code.

3. What is command parsing with DCGs?

ChatGPT: The DCG rules (`command/1`) are used to define the syntax of commands, which enables Prolog to understand the input string commands and convert them into corresponding structured operations.

You define several `command/1` rules to specify the types of commands you want to support, each with their own specific format.

For example, "add destination" should be followed by the destination name, start date, end date, and budget. These rules define the structure of the command so that the parser knows how to understand the user input.

`parse_command/2` is designed to extract meaningful command information from user input. First, split the user input into individual words using `split_string`.

Then use `phrase/2` to match the DCG rules to convert the user input into a structured command that Prolog can handle (e.g. `add_dest(...)`), so that you can use the structure to perform the corresponding operation.

4. We asked GPT for instruct how to save txt and load txt. GPT helps to give instruction for error handling. Error handling is achieved by validating inputs and ensuring that only valid operations are performed. For instance, checking whether a destination exists before adding an expense.

TASK B: Minesweeper Game

1. **Game Logic and Dynamic Predicates:** The game logic for this simplified Minesweeper version is built using dynamic predicates that help keep track of the state of the game. The 6x6 grid is represented conceptually with coordinate pairs, and the mines are placed randomly across this grid using the **place_mines** predicate, which ensures no overlapping by recursively finding unique locations. The game uses **mine** to represent the locations of mines, **revealed** to track cells that the player has uncovered, and **flagged** to manage cells flagged as containing possible mines—all managed as dynamic predicates that can be updated during gameplay.
2. **Recursive Logic and Neighboring Count Calculation:** The game logic relies on recursive operations to handle critical functions such as calculating the number of adjacent mines and managing user interactions. The **count_adjacent_mines** predicate is used to determine how many mines surround a given cell. This involves using the **adjacent** predicate, which generates all neighboring cells through relative coordinate offsets, and **findall** to compile a list of neighboring mines. Additionally, user commands like **reveal** and **flag** employ recursive behavior to modify game state dynamically—revealing or flagging cells based on the current state while ensuring that all neighboring conditions are respected.
3. **Game Interaction and Display:** The interaction between the user and the game is managed through several commands, including revealing cells, flagging cells, and starting the game. The game display is managed through the **display_grid** predicate, which iterates over every cell in the grid and shows either the number of adjacent mines, flagged cells, or unrevealed cells depending on their state. To facilitate easier debugging, a special function called **display_mines_for_debugging** is provided to display the complete grid with all mine positions before the game starts. Additionally, the win condition is handled by the **check_win_condition** predicate, which checks if all mines have been correctly flagged, allowing the player to win the game once all mines are correctly identified.

4. Testing and Results:

```
Grid initialized with 6 mines.  
# # # # # #  
# # # # # #  
# # # # # #  
# # # # # #  
# # # # # #  
# # # # # #  
true.
```

```
3 ?- flag(2, 5).  
Cell flagged.  
# # # # # #  
# # # # F #  
# # # # # #  
# # # # # #  
# # # # # #  
# # # # # #  
true.
```

```
7 ?- flag(5, 3).  
Cell flagged.  
# # # # # #  
# # # # F #  
# # # F # #  
# # # # # F  
# F F # # #  
# # # # # #  
true.
```

```
8 ?- flag(6, 5).  
Cell flagged.  
Congratulations! You flagged all the mines and won the game!
```



```
4 ?- flag(3, 4).
Cell flagged.
# # # # #
# # # # F #
# # # F # #
# # # # #
# # # # #
# # # # #
# # # # #
true.
```

```
5 ?- flag(4, 6).
Cell flagged.
# # # # #
# # # # F #
# # # F # #
# # # # # F
# # # # #
# # # # #
# # # # #
true.
```

```
6 ?- flag(5, 2).
Cell flagged.
# # # # #
# # # # F #
# # # F # #
# # # # # F
# F # # #
# # # # #
```

5. **Reference:** The method `check_win_condition` is constructed with the help of ChatGPT.