Imperial College London – Department of Computing

MSc in Advanced Computing
MSc in Computing Science (Specialism)
MRes in Advanced Computing

531: Prolog
Laboratory Test (ASSESSED)

THREE HOURS 15 MINUTES

INSTRUCTIONS

Answer *ALL FOUR* questions.

You will have **15 minutes** to read through the questions and make notes before you start work. You will have **3 hours** from the time you start working.

Add your answers to the provided file `lexis.pl`. You will find

- `lexis.pl`,
- `lexis.pdf` (this document)
- `support.pl`, a file containing test examples and useful programs,

in your Lexis home directory (`/exam`). If any of these files is missing, please alert one of the invigilators.

Save your work regularly. This is all that is required to submit your work. The final state of the file `lexis.pl` will be your submission. You are free to add to the examples in `support.pl` for testing purposes, but *DO NOT* include any part of your answers in this file. It does not form part of your submission and it will be discarded.

You may use any *BUILT-IN* Sicstus predicate. You may *NOT* use any Sicstus libraries unless stated. Ensure that your solution *COMPILES and EXECUTES* without errors on the Linux Sicstus implementation.

This question paper consists of **nine** printed pages, including this page.

Quoridor

All the questions in this test are concerned with the game *Quoridor*. You have been provided with a partial implementation of the game. You will complete the game by providing solutions to the questions below. Save all your work in the file named `lexis.pl`.

Quoridor is played by two players on a 9×9 board. Each player has a single piece to move, called a *pawn*, and the aim of the game is simply to cross the board from one side to the other. However, players are also issued with ten *fences* with which to block their opponent's progress.

Rules

1. Player 1 (◯) and Player 2 (×) start on opposite sides of the board, both in column e (see Fig. 1). Each player has ten fences. Player 1 takes the first turn, and the winner is the first player whose pawn reaches the opposite side of the board.

2. In each turn a player can either

   - move their pawn, or
   - add *one* of their fences to the board.

3. A fence placed on the board blocks the path of all pawns. Fences cannot be jumped.

4. In one turn a pawn can move to an *adjacent* empty square, or if the opposing pawn is occupying an adjacent square, *jump* that pawn. One square is adjacent to another if it is a horizontal or vertical (but *not* diagonal) neighbour.

5. A pawn that jumps its opponent will normally land directly behind it (e.g. in Fig. 2 ◯ could jump × and land at f7). If (and *only* if) such a linear jump is blocked because the opponent has a fence immediately behind them, the jumping pawn can make an L-shaped jump, and land in one of the squares adjacent to the jumped pawn. In Fig. 2 × cannot jump to f4, so it can jump to e5 instead. Note that the jump to g5 is blocked by the vertical fence.

6. A fence is two squares in length and is placed between squares (along the grid lines). Fences cannot be placed on top of one another or cross each other. Fences must align with the grid — they cannot partially obstruct a square. Thus each added fence divides a group of four squares in half, either horizontally or vertically.

7. A player may *not* add a fence anywhere that eliminates all remaining ways for their opponent to reach their target side of the board.

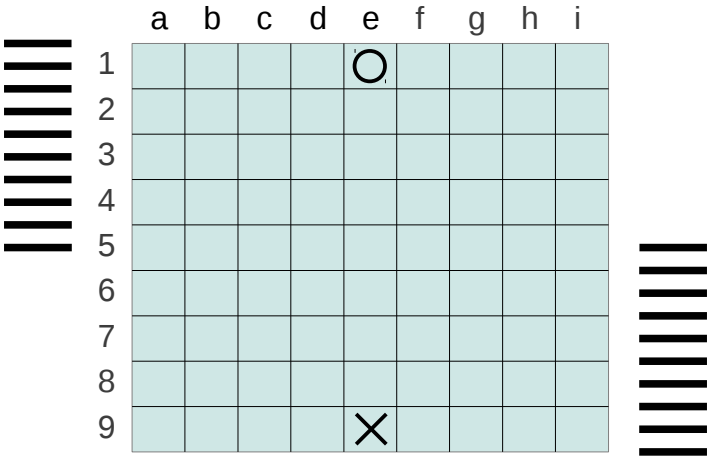

Figure 1: The start of the game. The players' remaining fences are shown beside the board.

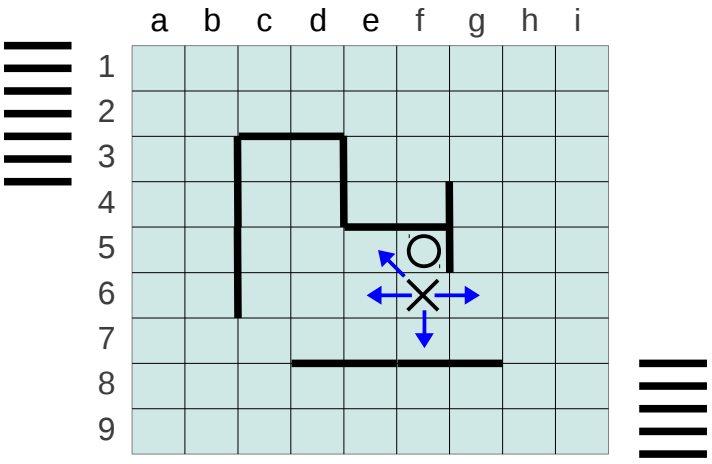

Figure 2: A game in progress. The four possible moves for the × pawn are shown by the arrows. If this was the turn of ◯, they could move left to e5 or jump × to f7.

Representing the Game

The internal, Prolog representations of the elements of the game are as follows.

- A player state is one of

  ```
  p1(X1,Y1,Fences1)
  p2(X2,Y2,Fences2)
  ```

  where: `X1,Y1` (resp. `X2,Y2`) are integers representing the position of Player 1 (resp. Player 2); and `Fences1` (resp. `Fences2`), an integer, is the number of fences Player 1 (resp. Player 2) still has to play. Squares are designated by `X` and `Y` integer values rather than letters and integers, so e1 is represented by `(5,1)`.

- A fence on the board is a term `(X,Y,Dir)`. Each such fence divides a group (a large square) of four board squares in half. The `X` and `Y` coordinates given in the fence term identify the top left board square of this group, and `Dir` is either `h` for horizontal or `v` for vertical. So, `(4,3,v)` is a vertical fence dividing d3 and d4 from e3 and e4.

- A board state is a list `[P1State, P2State, Fences]`, where `P1State` and `P2State` are player states and `Fences` is a list of fence terms. For example, the board in Fig. 3 is:

  ```
  [p1(5,1,8), p2(3,7,9), [(4,3,v),(4,7,h),(6,7,h)]]
  ```

- A move is either a term `(X,Y)`, denoting a pawn that moves to square `(X,Y)` or a fence term `(X,Y,Dir)` denoting the fence being added.

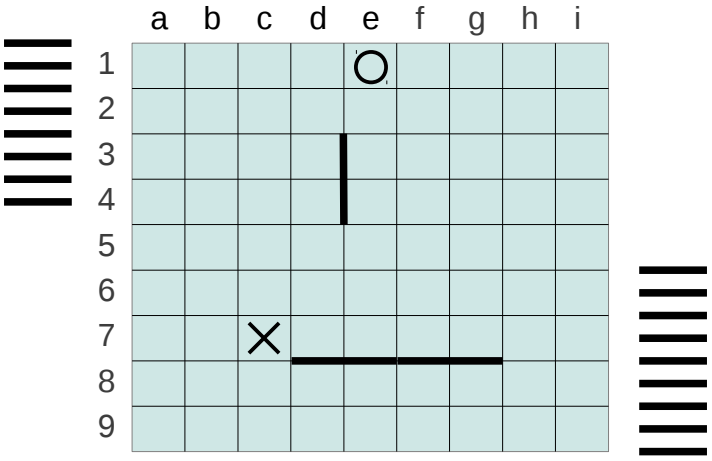

Figure 3: A board state.

Support File

The file `support.pl` contains some example states for testing, and provides definitions of:

- `show_board(State)` prints a board corresponding to the given `State` to standard output, where `State` is a valid board state (see 'Representing the Game'). This predicate will be used within your game, and as you wish for testing purposes.

- `v_fence_at(X,Y,Fs)` given integers `X` and `Y` and the list of fences `Fs`, succeeds if there is a vertical fence between `(X,Y)` and `(X+1,Y)`.
- `h_fence_at(X,Y,Fs)` given integers `X` and `Y` and the list of fences `Fs`, succeeds if there is a horizontal fence between `(X,Y)` and `(X,Y+1)`.
- `select_move(Player,Name,State,Move)` prompts `Player` for their next move, verifies the input is a legal move, and returns it as `Move`. The `select_move` program is only partially implemented. The answers to Questions 1–3 will complete it.

The example states can all be printed with `?- print_examples.` after consulting the file.

Question 1 (5%)

Players use a 'human readable' syntax to express moves. A move is either a string xy, e.g."e2", to move the player's pawn to square xy, or a string xyd, e.g. "d3v", to add a fence. In both cases x is a character in the range "a" $\leq x \leq$ "i", and y is a character in the range "1" $\leq y \leq$ "9". In a fence move d is either "h" or "v". This notation follows the same convention as the Prolog representation, so a move "d3v" adds a vertical fence dividing $d3$ and $d4$ from $e3$ and $e4$.

Write Prolog DCG rules for `read_move(-Move)`. Given a string `Str` corresponding to one of the human readable forms for a move described above, a call `phrase(read_move(Move), Str)` should return `Move`, the Prolog representation of `Str`. For example, the following queries should return `Move` as shown:

```
| ?- phrase(read_move(Move), "e2").
Move = (5,2) ? ;
no

| ?- phrase(read_move(Move), "d3v").
Move = (4,3,v) ? ;
no
```

Remember, Prolog strings are lists of character codes, so `"a1" == [97,49]` is true.

Question 2 (50%)

(a) (4 marks)

Write a Prolog program for `in_range(?N,+Min,+Max)` which, given integers `Min` and `Max`, succeeds if `N` is an integer in the range `Min` $\leq$ `N` $\leq$ `Max`. So, the query `?-in_range(N,2,4)` should generate the following output:

```
| ?- in_range(N,2,4).
N = 2 ? ;
N = 3 ? ;
N = 4 ? ;
no
```

If you wish to skip this part, uncomment the provided library solution.

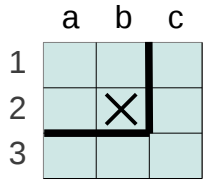

Figure 4: It's a trap! This state is, in fact, legal because × still has access to row 1.

(b) (12 marks)

Write a Prolog program for `fence_space(+Fences,?Space)` which, given the list `Fences` of fences currently on the board, returns a triple `Space` representing a fence that could be added. Ignore the positions of the pawns and whether they would be able to complete the game. If `Space` is a variable then all answers should be generated on backtracking. Thus, given the board in Fig. 3, the query

```
| ?- fence_space([(4,3,v),(4,7,h),(6,7,h)], Space).
```

should return answers including `Space = (1,1,h)` and `Space = (1,1,v)`, but not `Space = (4,3,h)` or `Space = (3,7,h)` since fences cannot be placed on top of one another.

(c) (30 marks)

Write a Prolog program for `reachable(+From, +Fences, ?Reachable)` which, given a pawn location `From`, where `From` is `(X,Y)`, and a list `Fences` of fence triples, returns a square `Reachable`, where `Reachable = (X1,Y1)` if and only if a pawn at `(X,Y)` can reach `(X1,Y1)`, given unlimited moves and ignoring the position of the other pawn. All answers should be generated on backtracking.

So, given the (unlikely!) board state in Fig. 4, the following query generates the squares reachable by Player 2:

```
| ?- reachable((2,2),[(1,2,h),(2,1,v), ...],Reachable).
Reachable = (2,2) ? ;
Reachable = (1,2) ? ;
Reachable = (1,1) ? ;
Reachable = (2,1) ? ;
no
```

(the order of the answers is not important).

As you may have realised, this is a graph search problem. Since the graph is cyclic you will need to avoid looping. Your program will need to maintain two sets, which you should implement using lists. The first set will contain *all* squares discovered by the search so far. The second set will contain squares that have been discovered and not yet used to expand the search. When a square is used to discover new reachable squares it should be removed from this second set.

HINT — You do not need to consider jumps (actually, you cannot). You may like to make use of the predicates `v_fence_at(X,Y,Fences)` and `h_fence_at(X,Y,Fences)` defined in `support.pl`.

(d) (4 marks)

Write a Prolog program `fence_move(+NumFs,+Fs,+OppSquare,+OppTarget,-NewF)` which, given `NumFs`, the number of fences a player has left to play, the list `Fs` of fences on the board, the pair `OppSquare = (OppX,OppY)` defining the position of the opponent's pawn, and `OppTarget`, the row the opponent is trying to reach, returns a fence `NewF = (X,Y,Dir)` that the player can add according to the rules. When `NewF` is a variable all answers should be generated on backtracking.

You will need to use your answers for parts 2 (a–c). If you did not complete these programs you can use the incomplete, or "stub", versions provided to test your answer to part (d).

Question 3 (30%)

Write a Prolog program `pawn_move(+Opp,+Fences,+From,?To)` which, given pairs `Opp = (OppX,OppY)` and `From = (X,Y)` defining the current locations of the two pawns, and a list `Fences = [(F1X,F1Y,F1Dir),...]` of fences currently on the board, returns a square `To = (X1,Y1)` that the pawn at `(X,Y)` can move to in a single turn. For example, taking the board shown in Fig. 2, the query `?- pawn_move((6,5),[(3,2,h),(2,3,v), ...],(6,6),To).` should generate the following output (the order of the answers is not important).

```
| ?- pawn_move((6,5),[(3,2,h),(2,3,v), ...],(6,6),To).
To = (5,5) ? ;
To = (5,6) ? ;
To = (7,6) ? ;
To = (6,7) ? ;
no
```

Question 4 (15%)

The final stage is to implement the code to combine the players' moves into a game.

(a) (2 marks)

Write a Prolog program for `game_over(+State)`, where `State` is a board state. This should succeed if `State` represents a board where one of the players has won the game. You can assume, without checking, that `State` is a valid, well-formed board.

(b) (8 marks)

Write a Prolog program for `next_state(+S,+Player,+Move,-S1)`. Given a legal board state `S`, `next_state` returns the state `S1` that results when the given `Player` (either `p1` or `p2`) makes the given legal `Move` (one of `(X,Y)` or `(X,Y,Dir)`). For example, given the board shown in Fig. 3, your program should return the following answers for the queries shown

```
| ?- next_state([p1(5,1,8),p2(3,7,9),[(4,3,v),...]], p1, (5,2), S1).
S1 = [p1(5,2,8),p2(3,7,9),[(4,3,v),...]] ? ;
```

```
no
| ?- next_state([p1(5,1,8),p2(3,7,9),[(4,3,v),...]], p2, (4,2,h), S1).
S1 = [p1(5,1,8),p2(3,7,9),[(4,2,h),(4,3,v),...]] ? ;
no
```

You can assume, without checking, that `S` is a valid state and that `Move` is an allowable move.

(c) (5 marks)

Define the predicate `play(+Player,+Name,+OppName,+State,-Winner)`. Given `Player` (either `p1` or `p2`) identifying the next player to take a turn, their name `Name`, their opponent's name `OppName`, and the current board `State`, `play/5` returns the winner's name as `Winner`. That is, `Winner` is one of `Name` or `OppName`.

If the game has already been won in `State`, then the program should determine the winner and return. If the game is not yet over, then `play` should carry out the next turn and then proceed with the rest of the game. To carry out a turn the program should use `select_move/4` (defined in `support.pl`) to get the next move for `Player`. This move should then be made, and the new board state should be displayed before continuing.

The program `select_move(+Player,+Name,+State,-Move)` prompts `Name` for the next move for `Player` (`p1` or `p2`) and translates the move from human readable syntax to its Prolog representation. If the move is legal it is returned in its Prolog form as `Move`, otherwise the player is prompted to try again. This program depends on your answers to Questions 1–3.

Submission

To answer the questions add new clauses to the supplied file `lexis.pl`.

Your edited versions should contain all the code you want to submit as your answer to the given questions, including any comments you regard as necessary to justify your solutions. You can include comments to outline your intended method if you cannot get your programs to work fully.

You are not required to do anything else; the file will be taken directly from your working space once you have logged out from your session. Any other files you make as copies or backups will be ignored. The support files will also be ignored. (Your answers will be tested on a different set of examples.)

Ensure that your submitted file *COMPILES WITHOUT ERRORS* on the Linux Sicstus implementation.

Do not use any of the Sicstus libraries unless stated in the question.