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 *BOTH* 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 useful programs,
- `testcases.pl`, `run.pl` and `Makefile` which allow you to run unit tests

in the `Lexis` directory inside your home directory. 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 tests in `testcases.pl`, but *DO NOT* include any part of your answers in this or any file other than `lexis.pl`. The other files do not form part of your submission and will be discarded.

You may use any *BUILT-IN* Sicstus predicate. You may *NOT* use any Sicstus libraries unless stated.

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

Testing

A set of unit tests for each part of each question has been provided in `testcases.pl`. To run these tests, open a new terminal and type `make`. The tests that run are controlled by the `run.pl` file. Initially no tests will run. To run a set of tests uncomment the relevant line in `run.pl`. The tests include the examples in the questions and some other simple cases. They have been provided to save you from manually running these obvious queries. They are NOT a comprehensive set of tests. You can add your own test cases to `testcases.pl` if you wish to run them via `make`. However, you are free to use any testing method, and `testcases.pl` will not be marked.

Question 1 (*50%, 25 marks*)

This question requires you to write programs that operate on a class of algebraic expressions. The expressions are based on the familiar Prolog arithmetic expressions, but can also contain the atoms `x`, `y` and `z`, to represent algebraic variables. So, in this question an expression is a *ground* Prolog term in one of the following forms:

- C
- V
- `-`(E)
- (E_1) `+` (E_2)
- (E_1) `*` (E_2)
- `sin(`E`)`
- `cos(`E`)`

where C is a number (float); V is one of `x`, `y` and `z`; E, E_1 and E_2 are expressions; `-`, `sin` and `cos` are unary Prolog operators; and `+`, `*` are binary Prolog operators. The operators represent the same functions they represent in Prolog. For example, the term

```
2.0 * sin(x + -5.0)
```

is such an expression, representing $2\sin(x-5)$.

Part (a) (6 marks)

Write a program `eval(+E, +Env, -V)`. Given the expression `E`, and the list `Env` of pairs `(Var, Val)` giving the value `Val` (a float) of each algebraic variable `Var` in `E`, the call `eval(E, Env, V)` should return the float `V` such that `E` evaluates to `V` when its variables take the values in `Env`. For example, the following queries should generate the answers shown:

```
| ?- eval(x + 3.0, [(x,5.0)], V).
V = 8.0 ? ;
no
| ?- eval((x + -2.0) * 2.0, [(x,5.0)], V).
V = 6.0 ? ;
no
```

You can assume without checking that `E` and `Env` are as described above.
NOTE. The Prolog built-in predicate `float/1` succeeds only if its argument is a float.

Part (b) (6 marks)

The binary operators `+` and `*` are both *commutative*, meaning the order of their operands is not important: $A + B = B + A$ and $A * B = B * A$. Write a program `commutes(+E1, +E2)` that compares expressions and succeeds if they are the same, ignoring the ordering of these commutative expressions. For example:

```
| ?- commutes(x+1.0, 1.0+x).
yes
| ?- commutes(x+1.0, x+1.0).
yes
| ?- commutes(sin(2.0*y), sin(y*2.0)).
yes
| ?- commutes(3.0+1.0, 2.0+2.0).
no
```

Part (c) (8 marks)

Write a program `diff(+E, +V, -D)` to differentiate expressions. Given an expression `E`, and an algebraic variable `V`, the call `diff(E, V, D)` should return the expression `D` representing

$$\frac{d\,\mathtt{E}}{d\,\mathtt{V}} \quad ,$$

the derivative of `E` with respect to `V`. The derivatives of each type of expression are:

$$\begin{aligned}
\frac{d}{dx}c &= 0.0 \\
\frac{d}{dx}x &= 1.0 \\
\frac{d}{dx}y &= 0.0, \qquad \text{when } y \neq x \\
\frac{d}{dx}(E_1 + E_2) &= \frac{d}{dx}E_1 + \frac{d}{dx}E_2 \\
\frac{d}{dx}(E_1 \times E_2) &= E_1 \times \frac{d}{dx}E_2 + E_2 \times \frac{d}{dx}E_1 \\
\frac{d}{dx}\sin(E) &= \cos(E) \times \frac{d}{dx}E \\
\frac{d}{dx}\cos(E) &= -\sin(E) \times \frac{d}{dx}E
\end{aligned}$$

where x and y are variables, E, E_1 and E_2 are expressions and c denotes a constant. For example, the following queries should produce the answers shown.

```
| ?- diff(x, x, D).
D = 1.0 ? ;
no
| ?- diff(x+x, x, D).
D = 1.0+1.0 ? ;
no
```

You can assume without checking that `E` and `V` are as described above. You should use the provided program `simplify/2` to simplify the derivative `D`. A call `simplify(E1, E2)` will attempt to simplify the expression `E1`, by removing occurrences of $+0.0$, $\times 1.0$ etc., returning the expression

`E2`. (If `E1` cannot be simplified then it is returned unchanged.) So, the following queries should produce the answers shown below.

```
| ?- diff(2.0*x, x, D).
D = 2.0 ? ;
no
| ?- diff(x*x, x, D).
D = x+x ? ;
no
```

NOTE Where the derivative is an expression like `2.0*x`, you do NOT need to generate other equivalent answers (in this case `x*2.0`) on backtracking. Any one of the possible alternatives will be considered correct.

Part (d) (5 marks)

A function $f(x)$ can be expressed as an infinite polynomial called a *Maclaurin series* [1]:

$$\begin{aligned} f(x) &= f(0) + f^{(1)}(0)x + \frac{f^{(2)}(0)}{2!}x^2 + \frac{f^{(3)}(0)}{3!}x^3 + \dots \\ &= \sum_{i=0}^{\infty} \frac{f^{(i)}(0)}{i!}x^i \end{aligned}$$

where $n! = 1 \times 2 \times \cdots \times n$; and $f^{(n)}(x)$ is the nth derivative of the function, so $f(x) = f^{(0)}(x)$ and

$$f^{(n)}(x) = \frac{d}{dx}(f^{(n-1)}(x)), \quad (n > 0).$$

Write a program `maclaurin(+E, +X, +N, -V)` which uses (the beginning of) the Maclaurin series to approximate the value of a function $f(x)$ for a given x. Given the expression `E` containing no variable other than `x`, the number `X`, and the positive integer `N`, the call `maclaurin(E, X, N, V)` should return the float `V` such that `V` is the sum of the first `N` terms of the Maclaurin series for `E` when `x=X`. For example, the queries to find the value of $f(x) = x$ when $x = 2$ using two Maclaurin terms, and the value of $f(x) = sin(x)$ when $x = 2$ using four and then thirty Maclaurin terms, and their answers are:

```
| ?- maclaurin(x, 2.0, 2, V).
V = 2.0 ? ;
no
| ?- maclaurin(sin(x), 2.0, 4, V).
V = 0.6666666666666667 ? ;
no
| ?- maclaurin(sin(x), 2.0, 30, V).
V = 0.9092974268256817 ? ;
no
```

You can assume without checking that `E`, `X` and `N` are as described above.

[1] Actually, this is only true if the series is *convergent* for $f(x)$. You can assume this is the case here.

Question 2 (*50%, 25 marks*)

In this question you will write a Prolog program to solve a puzzle. In the puzzle there are seven squares in a line and six *tiles*, three black and three white, that start in the positions shown in Figure 1. The aim of the puzzle is to get all the white tiles to the left of all the black tiles, by moving tiles into the empty square. The final position of the empty square does not matter. A tile can be moved if there are *at most* two other tiles between it and the empty square. This is counted as one *move*, regardless of the number of squares the tile moves.

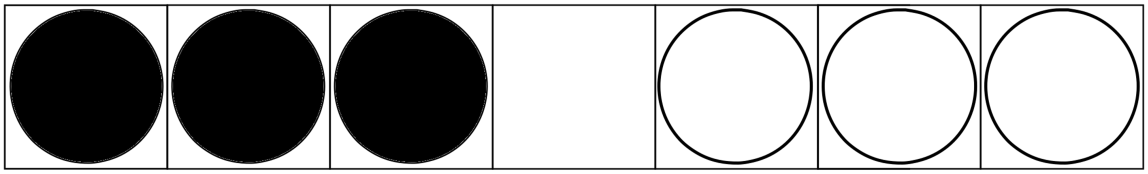

Figure 1: The starting position.

In the Prolog program, a *position* is a list of 7 terms, `b` for a black tile, `w` for a white tile, and `e` for the empty square. So, the starting position is `[b,b,b,e,w,w,w]`.

You will solve the puzzle by searching possible states, beginning at the starting position (Fig. 1). The first part of the program has been provided in `support.pl`. You need to define the predicates described in parts (a–d) to complete the program. The finished program can be run by calling `tiles` and should output a solution like:

```
| ?- tiles.
[b,b,b,e,w,w,w]-9
[b,b,b,w,w,e,w]-9
[b,b,e,w,w,b,w]-7
[e,b,b,w,w,b,w]-7
[w,b,b,e,w,b,w]-5
[w,b,b,w,w,b,e]-4
[w,b,b,w,w,e,b]-4
[w,b,e,w,w,b,b]-2
[w,b,w,w,e,b,b]-2
[w,e,w,w,b,b,b]-0
yes
```

Part (a) (4 marks)

The search will be guided by a *heuristic function* $h(Pos)$, which estimates the number of moves needed to solve the puzzle from position Pos by summing the number of black tiles found to the left of each white tile [2]. So, the value of $h(Pos)$ for the starting position is 9, because there are 3 black tiles to the left of each white tile.

Write a program `b_to_left(+Pos, -H)`. Given a position `Pos`, the call `b_to_left(Pos, H)` should return the integer `H` such that `H` is the sum of the number of black tiles found to the left of each

[2] This heuristic sometimes *over-estimates* the number of moves required, meaning the search is "A", not "A*".

white tile in `Pos`. You can assume without checking that `Pos` is a valid position. For example, the following query returns `H = 9`:

```
| ?- b_to_left([b,b,b,e,w,w,w], H).
H = 9 ? ;
no
```

Part (b) (8 marks)

Write a program `move(+From, -To)`. Given a position `From`, the call `move(From, To)` should return `To` if and only if `To` is a position that results when one of the tiles in `From` is moved into the empty space, jumping *at most* two tiles. You can assume without checking that `From` is a valid position. For example, the following query returns four answers for `To`:

```
| ?- move([b,e,b,b,w,w,w], To).
To = [e,b,b,b,w,w,w] ? ;
To = [b,b,e,b,w,w,w] ? ;
To = [b,b,b,e,w,w,w] ? ;
To = [b,w,b,b,e,w,w] ? ;
no
```

Part (c) (9 marks)

The search is controlled by a list of search *nodes* called an *agenda*. A search node is a term `n(F,G,Pos,ID,PID)`, containing a position `Pos` plus the following other information.

- `G` is the number of moves it has taken to reach `Pos` from the starting position.
- `F` is `G` + h(`Pos`). This is called the *evaluation function* and gives an estimate of the total number of moves needed to solve the puzzle.
- `ID` is an integer that uniquely identifies this node.
- `PID` is the unique id of this node's *parent* (see below) or `none` if it has no parent.

The agenda is kept in ascending order by `F`, the evaluation function. The first node on the agenda is always *visited* next, making this a *best-first* search. The visited node is removed from the agenda. If it is a goal state the search is over. Otherwise, all the *children* of its position are found and added to the agenda, to be visited later. The children are nodes containing positions that result from making any move from the visited *parent* node's position. Visited nodes and their parents are remembered, enabling the program to trace the path back to the start once the goal is reached.

Write a program `search_agenda(+Agenda, -Visited, -Final)` that performs a best-first search. Given a list `Agenda` of nodes in ascending order by evaluation function, the call `search_agenda(Agenda, Visited, Final)` should return a node `Final` containing a goal state that can be reached from one of the nodes in `Agenda`, and the list `Visited` containing all nodes, other than `Final`, visited during the search. For example, the first answer to the following query, with an agenda containing one node one move away from a goal state, is shown below:

```
| ?- search_agenda([n(4, 3, [w,w,b,w,e,b,b], _, 5)], Visited, Final).
Visited = [n(4,3,[w,w,b,w,e,b,b],0,5)],
Final = n(4,4,[w,w,e,w,b,b,b],_A,0) ? ;
yes
```

You can assume without checking that `Agenda` is as described.

NOTE:

1. A node should be given a unique integer id when it is visited (removed from the agenda). Before this its id is undefined. So, the node containing the starting position is initially `n(9,0,[b,b,b,e,w,w,w],_,none)`.

2. A *sorted* list of child nodes can be efficiently added to the agenda by merging the two lists. Full credit will only be given to programs using this method, or something equally efficient.

The file `support.pl` defines the following predicates for interacting with node terms. These abstract the `n/5` terms and make it possible to create and access nodes without needing to use an `n/5` pattern directly. It is up to you whether you use them or not. Given a position `Pos`, an integer `G`, an integer `H` $= h($`Pos`$)$, and a parent id `PID`, the call:

```
make_node(G, H, Pos, PID, N)
```

creates a new node `N`. And given a node `N`, then its data can be accessed by calling:

```
node_f(N, GVal)
node_g(N, GVal)
node_h(N, HVal)
node_pos(N, NPos)
node_id(N, NID)
node_pid(N, NPID)
```

Part (d) (4 marks)

The final part of the program is to extract the solution: the sequence of positions from the starting position to the final one (when actually solving the puzzle this would be a goal position) using the information in the visited nodes. Since the search will have visited other nodes too, it will be necessary to use the node ids to find the right sequence.

Write a program `trace_moves(+Final, +Visited, -Seq)` to extract a sequence of moves from the visited search nodes. Given a node `Final`, and a list `Visited` of nodes that includes a path from the starting position to a parent of `Final`, the call `trace_moves(Final, Visited, Seq)` should return the list `Seq` of terms `[T1,...,Tn]` each of the form `Pos-H`, where `Pos` is a position and `H` is the value of $h($`Pos`$)$, such that `T1` contains the starting position, `Tn` contains the position in `Final`, and `Ti` contains the position in the parent of the node containing the position in `T(i+1)`, for `i=1 to n-1`. For example, the following query, in which `Final` is a child of the starting position node, returns `Seq` as shown:

```
| ?- trace_moves(n(10, 1, [b,b,e,b,w,w,w], _, 0),
     [n(9, 0, [b,b,b,e,w,w,w], 0, none)], Seq).
Seq = [[b,b,b,e,w,w,w]-9,[b,b,e,b,w,w,w]-9] ? ;
no
```

You can assume without checking that `Final` and `Visited` are as described, and that `Visited` is ordered in the same way as the list generated by your `search_agenda` program.

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.