# COMP9414/9814 Artificial Intelligence

# Session 1, 2016

### Assignment 1 - Prolog Programming

Due: Friday 8 April, 11:59pm
Marks: 10% of final assessment for COMP9414 Artificial Intelligence, or 10% of final assessment for COMP9814 Extended Artificial Intelligence

---

### Specification

In this assignment, you are to write Prolog procedures to perform some list and tree operations. The aim of the assignment is to give you experience with typical Prolog programming techniques.

At the start of your program, place a comment containing **your full name, student number and assignment name**. You may add additional information like the date the program was completed, etc. if you wish.

At the start of each Prolog predicate that you write, write a comment describing the overall function of the predicate.

Advice on the use of comments and meaningful identifiers in Prolog can be found under comments in the Prolog Dictionary.

---

### Testing Your Code

A significant part of completing this assignment will be testing the code you write to make sure that it works correctly. To do this, you will need to design test cases that exercise every part of the code.

You should pay particular attention to "boundary cases", that is, what happens when the data you are testing with is very small, or in some way special. For example:

- What happens when the list you input has no members, or only one member?
- Does you code work for lists with both even and odd numbers of members?
- Does your code work for negative numbers?

Note: not all of these matter in all cases, so for example with `sqrt_table`, negative numbers don't have square roots, so it doesn't make sense to ask whether your code works with negative numbers.

With each question, some example test data are provided to clarify what the code is intended to do. You need to design *further* test data. Testing, and designing test cases, is part of the total programming task.

---

### COMP9814-only Questions

Questions marked **COMP9814 only** need only be answered by students enrolled in COMP9814. (Students enrolled in COMP9414 are welcome and encouraged to solve these problems but their solutions will not be assessed.)

**It is important to use <u>exactly</u> the names given below for your predicates, otherwise the automated testing procedure will not be able to find your predicates and you will lose marks. Even the capitalisation of your predicate names must be as given below.**

---

1. Write a predicate `sumsq_even(Numbers, Sum)` that sums the squares of only the even numbers in a list of integers. *Example:*

```
?- sumsq_even([1,3,5,2,-4,6,8,-7], Sum).

Sum = 120 ;

false.
```

Note that it is the element of the list, not its position, that should be tested for oddness. (The example computes `2*2 + (-4)*(-4) + 6*6 + 8*8`.) Think carefully about how the predicate should behave on the empty list — should it fail or is there a reasonable value that `Sum` can be bound to?

In order to decide whether a number is even or odd, you can use the built-in Prolog operator `N mod M`, which computes the remainder after dividing the whole number `N` by the whole number `M`. Thus a number `N` is even if the goal `0 is N mod 2` succeeds. Remember that arithmetic expressions like `X + 1` and `N mod M` are only evaluated, in Prolog, if they appear *after*the `is` operator. So `0 is N mod 2` works, but `N mod 2 is 0` doesn't work.

---

2. For the purposes of the examples in this question, assume that the following facts have been loaded into Prolog:

```
likes(mary, apple).
likes(mary, pear).
likes(mary, grapes).
likes(tim, mango).
likes(tim, apple).
likes(jane, apple).
likes(jane, mango).
```

   **NOTE:** do not include these in your solution file.

   Write a predicate `all_like(What, List)` that succeeds if `List` is empty or contains only persons that like `What`, according to the predicate `likes(Who, What)`. *Example:*

```
?- all_like(apple,[mary, tim]).

true ;
false.

?- all_like(grapes,[mary, tim]).

false.

?- all_like(pear,[]).

true ;
false.
```

   **COMP9814 only:** Write a predicate `all_like_find(+Object, -List)` that binds `List` to the list of all persons who like `Object`.

```
?- all_like_find(mango, Who).

Who = [tim, jane] ;

false.
```

   Note that your `all_like` predicate will be tested with different `likes(Who, What)` facts to those in the examples.

---

3. write a predicate `sqrt_table(N, M, Result)` that binds `Result` to the list of pairs consisting of a number and its square root, from `N` down to `M`, where `N` and `M` are non-negative integers, and `N >= M`. For example:

```
sqrt_table(7, 4, Result).

Result = [[7, 2.64575], [6, 2.44949], [5, 2.23607], [4, 2.0]] ;

false.
```

Note that the Prolog built-in function sqrt computes square roots, and needs to be evaluated using `is` to actually compute the square root:

```
?- X is sqrt(2).

X = 1.41421 ;

false.
?- X = sqrt(2).

X = sqrt(2) ;

false.
```

---

**COMP9814 only:** write a predicate `function_table(+N, +M, +Function, -Result)` that binds `Result` to the list of pairs consisting of a number X and Function(X), from `N` down to `M`. For example:

```
?- function_table(7, 4, log, Result).

Result = [[7, 1.94591], [6, 1.79176], [5, 1.60944], [4, 1.38629]] ;

false.
```

`log` computes the natural logarithm of its argument.

Hint: look up `univ` or `=..` in the Prolog Dictionary.

---

4. Write a predicate `chop_down(List, NewList)` that binds `NewList` to `List` with all sequences of *successive decreasing* whole numbers replaced by the last number in the sequence. An example of successive decreasing whole numbers is: `22, 21, 20, 19`. (Note that the numbers have to be *successive* in the sense of decreasing by exactly 1 at each step.) For example:

```
?- chop_down([1, 3, 7, 6, 5, 10, 9], Result).
Result = [1, 3, 5, 9] ;
false.
?- chop_down([6, 4, 10, 9], Result).
Result = [6, 4, 9] ;
false.
```

In this example, the sequence `7, 6, 5` has been replaced by `5`, and `10, 9` has been replaced by `9`.

---

5. For this question we consider binary expression-trees whose leaves are either of the form `tree(empty, Num, empty)` where `Num` is a number,

or `tree(empty, z, empty)` in which case we will think of the letter `z` as a kind of "variable". Every tree is either a leaf or of the form `tree(L, Op, R)` where `L` and `R` are the left and right subtrees, and `Op` is one of the arithmetic operators `'+'`, `'-'`, `'*'`, `'/'` (signifying addition, subtraction, multiplication and division).

Write a predicate `tree_eval(Value, Tree, Eval)` that binds `Eval` to the result of evaluating the expression-tree `Tree`, with the variable `z` set equal to the specified `Value`. For example:

```
?- tree_eval(2, tree(tree(empty,z,empty),
                 '+',tree(tree(empty,1,empty),
                      '/',tree(empty,z,empty))), Eval).
Eval = 2.5 ;
false.

?- tree_eval(5, tree(tree(empty,z,empty),
                 '+',tree(tree(empty,1,empty),
                      '/',tree(empty,z,empty))), Eval).
Eval = 5.2 ;
false.
```

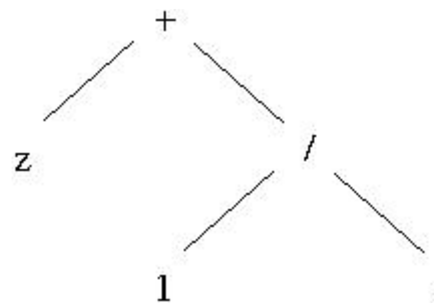
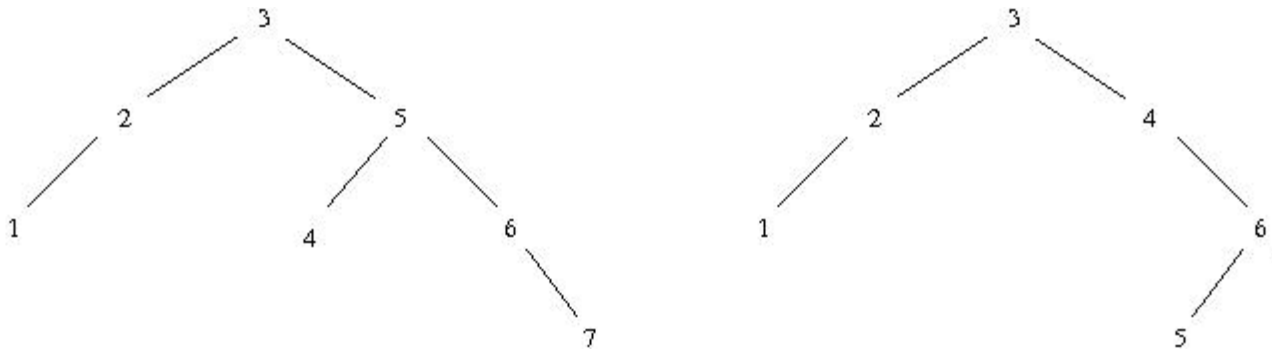



Illustration of the tree used in the example above.

---

**COMP9814 only:** The **height** of a binary tree is defined to be the number of nodes in the longest path from the root to a leaf. A binary tree is called **balanced** if, for every node in the tree, the height of its left and right subtree differ by no more than 1. For example, the tree on the left is balanced, with height 4, but the tree on the right is not balanced, because the left and right subtree of node 4 have heights 0 and 2, respectively.




**COMP9814 only:** Write a predicate `height_if_balanced(Tree, HiB)`, which takes a binary tree `Tree` and binds its second argument `HiB` to the height of the Tree, if it is balanced, or -1 if it is **not** balanced.

Trees are represented as either `empty` or `tree(L, Data, R)`, where `L` and `R` are the left and right subtrees. The `Data` in each node of the tree is irrelevant to this programming exercise.

You are free to copy the predicate `max(First, Second, Max)` from the lecture notes and use it in your program.

`Tree` must be instantiated at the time of the call. These examples use the trees shown above:

```
?- height_if_balanced(tree(tree(tree(empty,1,empty),2,empty),
                        3, tree(tree(empty,4,empty),
                             5,
tree(empty,6,tree(empty,7,empty)))),HiB).
HiB = 4 ;
false.

?- height_if_balanced(tree(tree(tree(empty,1,empty),2,empty),
                         3, tree(empty,4,tree(tree(empty,5,empty),
                                       6,
tree(empty,7,empty)))),HiB).
HiB = -1 ;
false.
```

---

## Testing

Your assignment will be tested by an automated testing system, and also read by a human marker. Marks will be allocated for test results, and for layout, comments, and comprehensibility.

Your code must work under the version of SWI Prolog used on the Linux machines in the UNSW School of Computer Science and Engineering. If you develop your code on any other platform, it is your responsibility to re-test and if necessary correct your code when you transfer it to a CSE Linux machine prior to submission.

---

### Submitting your assignment

Put the Prolog code for all problems into a single file for submission purposes.

**cs9414 students: To hand in**, log in to a School of CSE Linux workstation or server, make sure that your program is in the current working directory, and use the Unix command:

```
% give cs9414 prolog mycode.pl
```

where `mycode.pl` is replaced by the name of the file with your code in it.

**cs9814 students: To hand in**, log in to a School of CSE Linux workstation or server, make sure that your program is in the current working directory, and use the Unix command:

```
% give cs9414 extprolog mycode.pl
```

where `mycode.pl>` is replaced by the name of the file with your code in it. (Yes, that's `cs9414>` and `extprolog`).

**Please make sure your code works on CSE's Linux machines and generates no warnings. Remove all test code from your submission, including that for question 2. Make sure you have named your predicates correctly.**

You can submit as many times as you like - later submissions will overwrite earlier ones. You can check that your submission has been received by using the following command:

```
9414 classrun -check
```

The submission deadline is Friday 8 April, 11:59 pm.
15% penalty will be applied to the (maximum) mark for every 24 hours late after the deadline.

Questions relating to the project can be posted to the MessageBoard on the [COMP3411 Web page](#).

If you have a question that has not already been answered on the MessageBoard, you can email it to `blair@cse.unsw.edu.au`

### Plagiarism Policy

Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for any similar projects from previous years) and serious penalties will be applied, particularly in the case of repeat offences.

**DO NOT COPY FROM OTHERS; DO NOT ALLOW ANYONE TO SEE YOUR CODE**

Please refer to the Yellow Form, or the CSE Addendum to the UNSW Plagiarism Policy if you require further clarification on this matter.