# yz709-prolog-sup1



## Question 1

Compare and contrast imperative, functional, and declarative programming.

| Imperative programming | Functional programming | Declarative programming |
| --- | --- | --- |
| Focus on the steps of the execution, treat a program as statements that will change the state of the program, e.g., C++ | Treat a program as a set of mathematical functions and avoid state and mutable data, e.g., OCaml | Focus on what to execute, defines the program logic but not the steps of the detailed control flow, e.g., Prolog, SQL |

- Imperative and declarative programming are complementary paradigms. The attribute of referential transparency (i.e., no side-effect) of expressions differentiate between a declarative expression and an imperative one.
- Declarative programming have no loops, functional programming introduces function recursion based on the general declarative programming paradigm, and makes the function a first-class object (i.e., enable higher order functions, functions can appear in arguments as a type).

 Comments: 

# Question 2

## Prolog Basics

### 1.1 (Bookwork)

Specify the rules Prolog uses for unification

- **Unification** is the Prolog's fundamental operation. (1) Atoms unify with atoms if identical (e.g., $a = a, a \neq b, a \neq a(A)$); (2) Variables unify with anything; (3) Compound terms unify if same `functor/arity` and unified `arguments` (e.g., $tree(A, r) = tree(l, B)$, $tree(l, r) = A$)

💡 Comments: 

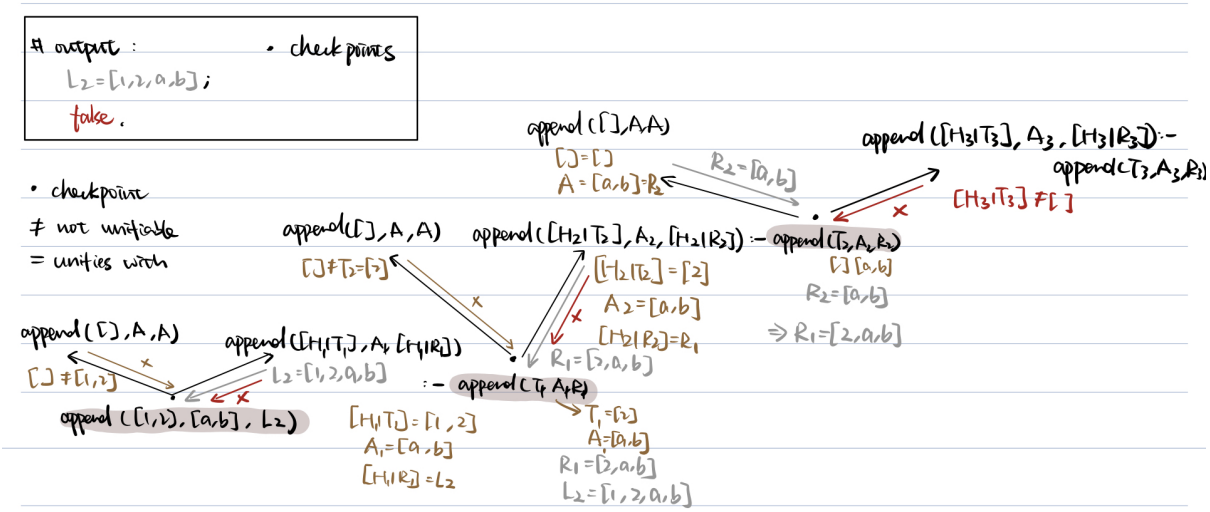
## Lists

### 4.3 (Shallow)

Consider the implementation of append given below. Explain how it should be used and draw out a search tree for a representative example.

```
append([],A,A).
append([H|T],A,[H|R]) :- append(T,A,R)
```

- `append(L,L1,L2)` will succeed if `L2` is a concatenation of `L` and `L1`, e.g., if `L = [1,2,3]`, `L1 = [a,b,c]`, and `L2 = [1,2,3,a,b,c]`, then it is true. Hence we need two input lists `L` and `L1` with a variable `L2`, and the results are stored in the output list `L2` once the program terminates.




Comments:

## 4.4 (Deeper)

You are given two implementations of member

```
member(H,[H|_]).
member(H,[_|T]) :- member(H,T).
% or, alternatively
member(X,Y) :- append(_,[X|_],Y).
```

1. Discuss how the second implementation compares with the use of 'partial application' from functional programming, and how it differs.
    - In the second implementation, all calls to `member/2` triggers a call to `append/3`, unlike partial application from functional programming, this does not require an additional argument to be passed.
    - The return type of the partial application function is a function of remaining arguments, and the types will change when we pass additional arguments to the function. However, for the second implementation above, the types remains as compound terms.
    - Partial application enables lazy evaluation, but in the second implementation, the evaluation will execute without delay.
2. If the two implementations are equivalent, should a programmer simply inline all calls to member with a call to append?

- From the performance perspective, both implementations require a stack space of $O(N)$ for exploring the whole search tree, since the built-in `append` is much less error-prone, we could inline all calls to member with a call to append.
- From the perspective of readability, the second implementation hides the details of the rules and requires the programers to understand the linkages between `member` relation and `append` relation, so we shouldn't.

3. If the two implementations are equivalent, what advantages does one form of expression have over the other?
   - Implementation with `append` have three list arguments, requires $2N$ space to store the list elements, but first `member` implementation have only one atom argument and one list argument requiring $N+1$ space to store the elements.
   - Implementation with `append` is a built-in function, hence much less error-prone for programmers to implement the `member` program.
   - The first `member` implementation is straight-forward and more intuitive, the second implementation with `append` requires an understanding of the usage of the `append` program.
4. Prove (informally) why the two are logically equivalent.
   - Use prove by rule induction, the base fact case returns true from both implementations; for the rule case, we can prove that the rule body clauses from the second implementation unifies with the rule body clauses of the first implementation.

member(H, [H|_]) = append([], [H|_], [H|_]) = true.

member(H, [_|T]) = append(_, [H|_], [_|T])
= append(T1, [H|_], T) where _ = [_|T1]
= member(H, T)

Comments:

## 4.6 (Deeper)

What does the following do and how does it work?

```
b(X,X) :- a(X).
b(X,Y) :- append(A,[H1,H2|B],X), H1 > H2, append(A,[H2,H1|B],X1), b(X1,Y).
```

- I am not sure what is `a(X)` in this case, but from the supervision worksheet, it should be:

  ```
  a([]).
  a([H|T]) :- a(T,H).
  a([],_).
  a([H|T],Prev) :- H >= Prev, a(T,H).
  ```

- With the above assumption, we can deduce that `b(X,Y)` succeeds if `a(Y)` succeeds and `X` is a permutation of `Y`, i.e., `Y` is an ordered list (ascending-order).

  ```
  % a(L) succeeds if all elements in L are greater than their previous ones
  % i.e., L is an ordered list
  a([]).
  a([H|T]) :- a(T,H).
  a([],_).
  a([H|T],Prev) :- H >= Prev, a(T,H).

  % b(X,Y) succeeds if a(Y) succeeds and X is a permutation of Y.
  % i.e., Y is an ordered list, X is a permutatio of Y
  b(X,X) :- a(X).
  b(X,Y) :- append(A,[H1,H2|B],X), H1 > H2, append(A,[H2,H1|B],X1), b(X1,Y).
  ```

- With two input lists $X$ and $Y$, the fact branch checks whether two lists are the same and if so, check `a(X)` (i.e., all elements in $X$ are greater than their previous ones).
- When the input lists can not be unified (i.e., are not the same), we jump into the rule branch,
- `b(X,Y)` may return multiple `true` because for there are multiple branches/ways that we could reorder `X` to `Y`.

💡 Comments: 🟨

# Arithmetic

## 5.4 (Deeper)

The `is` operator in Prolog evaluates arithmetic expressions. This builtin functionality can
also be modelled within Prolog's logical framework. Let the atom `z` represent the smallest number (this is zero) and the compound term `s(A)` represent the successor of A. For example `3 = s(s(s(z))).` Implement and test the following rules (note: you should not use is to do all of the arithmetical work):

`prim(A,B)` which is true if A is a number and B is its primitive representation

`plus(A,B,C)` which is true if C is A+B (all with primitive representations, A and B are both ground terms

`mult(A,B,C)` which is true if C is A*B (all with primitive representations, A and B are both ground terms)

```
% prim(A,B) succeeds if A is a number and B is its primitiev representation
prim(0,z).
prim(A,s(B)) :- prim(DecA,B), A is DecA + 1.
% plus(A,B,C) succeeds if C is A + B
% (all in primitive representations, A and B are both ground terms)
plus(A,B,C) :- prim(NA,A), prim(NB,B), prim(NC,C), NC is NA + NB.
% mult(A,B,C) succeeds if C is A * B
% (all in primitive representations, A and B are both ground terms)
mult(A,B,C) :- prim(NA,A), prim(NB,B), prim(NC,C), NC is NA * NB.

% testing:
?- prim(NA,s(z)).
NA = 1.

?- prim(5,B).
B = s(s(s(s(z)))) .

?- plus(s(z),s(z),s(s(z))).
true.

?- prim(1,One),prim(2,Two),plus(One,One,Two).
One = s(z),
Two = s(s(z)) .

?- prim(3,Three),prim(2,Two),mult(Three,Two,Six).
Three = s(s(s(z))),
```

```
Two = s(s(z)),
Six = s(s(s(s(s(s(z)))))) .
```

Comments:
Previously, I got `arguments are not sufficiently instantiated` in prim when I want to pass `prim(NA,s(z))`, because I used `DecA is A - 1` but at that point A is not initialised.

## 5.7 (Shallow)

Explain what each of these queries evaluate to, and explain why you got the evaluation you did.

1. `1 + 1 = 2`
2. `1 + 1 is 2`
3. `1 + 1 =:= 2`
4. `One + One = Two`
5. `prim(1, One), prim(2, Two), plus(One, One, Two)`

(This requires you to have implemented the `prim` and `plus` predicates described earlier.)

- (1) false, because +(1,1) is a compound term, it cannot unify with a constant 2
- (2) false, because the left hand side is a compound term not a constant or a variable, which is not allowed for the `is` operator
- (3) true, the operator `=:=` evaluates both LHS and RHS ground terms, then compare them, the LHS evaluates to 2 which unifies with RHS
- (4) true, because `+(One,One)` is a compound term with arity 2, it can be unified with a variable `Two`
- (5):
  - The operator `,` indicates `and`, so we have to evaluate the relations in turn, find values for `One` and `Two` such that all relations are true.
  - `prim(1,One)` evaluate to true if `One` is the primitive representation of number `1`, so we have `One = s(z)`; `prim(2, Two)` evaluates to true if `Two` is the primitive representation of number `2`, so we have `Two = s(s(z))`; then for `plus(One, One, Two)`, we have already initialised `One` and `Two`, and after evaluating `plus(s(z), s(z), s(s(z)))`, we can prove it is true. So output the

result, if we ask for more answers using `;`, then the program runs forever because backtracking algorithm cannot find another valid answer, but searches forever.

```
?- prim(1,One),prim(2,Two),plus(One,One,Two).
One = s(z),
Two = s(s(z)) ;
% runs forever
```

💡 Comments:

# Question 3(y2020p7q10)

In this question we will write a program to assign guests to seats around a dinner table. The table is rectangular with the same number of seats along both of the long sides. There are no seats along the short sides. Two guests sit *near* each other if they occupy adjacent seats on the same side of the table, or sit directly opposite one another. Each guest at the dinner table may express a preference to sit near to one or more other guests.

When answering this question you should ensure that each of your predicates has a comment giving a declarative reading of its behaviour and you should avoid unnecessary use of cut. Your solutions should not use any extra-logical predicates (such as assertz) and you should not assume the existence of any library predicates.

(*c*) Implement a predicate **`nextTo(A,B,Assigned)`** which is true if guest **`A`** is sitting adjacent-to or directly opposite guest **`B`** in the seating assignment **`Assigned`**.
[4 marks]

- Suppose we have a `seat(Guest, Left, Right)` for each position representing the left adjacent seat and right adjacent seat to that guest seat. Loop around the table anti-clockwisely, assuming each set faces the centre of the table, e.g., if the seat is at the corner, then its left or right adjacent seat should be the seat opposite to it.
- `Assigned` is a list of `seat(Guest, Left, Right)`

- Because of symmetry, if A sits at the left side of B then B sits at the right side of A, so we only need to write two facts rather than four facts.

```
% assume Assigned is a list of seat(guest, left, right)
% nextTo(A,B,Assigned) succeeds if A is sitting adjancet to
% or directly opposite B in the seating assignment Assigned
nextTo(A,B,[seat(A,B,_)|_]).
nextTo(A,B,[seat(A,_,B)|_]).
nextTo(A,B,[_|T]) := nextTo(A,B,T).
```

(*d*) Implement a predicate **satisfied(Assigned,Prefs)** which is true if **Assigned** is a seating assignment which meets all the preferences specified in **Prefs**. [3 marks]

- Since each one guest may specify one or more guests they would want to sit with, so we could have a compound term `(Guest, L)` where `L` is a list of guests `Guest` would like to sit with.
- `Pref` is a list of `(Guest, L)`

```
% satisfied(Assigned, Prefs) succeeds
% if Assigned satisfies all preferences indicated in Prefs
satisfied(_,[]).
satisfied(Assigned,[(A,[B|T1])|T2])
          :- nextTo(A,B,Assigned), satisfied(Assigned, [(A,T1)|T2]).
satisfied(Assigned,[(_,[])|T1]) :- satisfied(Assigned, T1).
```

Comments: