Examen L3 Miage - Programmation déclaratives - Session 1, Mai 26

June 3, 2026

1 Exercice 1, sur la base du projet

Pas de solution, je vous laisse les trouver pour un prochain projet, mais des commentaires sur les erreurs :

- Sur l'inversion des rôles dans l'historique, cela tient en 2 lignes et ne parle pas de stratégie ; c'est un programme classique, indépendant du projet, vous pourrez le retrouver sur le web sans difficulté.
- Cette inversion est nécessaire pour faire jouer une IA contre elle-même. La stratégie utilisée aura 2 coups à jouer, un coup normal, pour l'IA-Normal, comme dans le projet, en prenant comme base l'organisation de l'historique de base avec des coups en premier dans l'historique pour l'opposant, et des coups en second pour "elle". Pour l'IA-symétrique, il faudra qu'elle utilise l'historique inversé, car si l'IA-Symétrique doit remplacer l'humain, il faut que les coups de l'opposant soient en second et ses coups en premier,
  - il faut donc inverser les rôles humain-IA dans l'historique pour pouvoir faire jouer cette IA-symétrique avec les coups de l'humain comme support pour représenter ses coups à elle.
  - Rem. : en général, les IA comportent un peu de hasard, faire jouer une IA contre elle-même n'est donc pas stérile et n'aboutit pas toujours à des matchs nuls ;
  - dans le mini-projet, le hasard n'est pas prévu pour rendre les parties rejouables, faire jouer une IA contre elle-même, ainsi, ne sera donc pas très utile, mais pour faire jouer une IA contre une autre IA, il faut prévoir de faire cela aussi…
  - Cette seconde question se traduit en quelques lignes de code (pas beaucoup), deux appels à l'application de la stratégie, et un appel à l'inversion pour produire l'historique inversé.
- La troisième stratégie (démocratique : on joue le coup favori de la population des stratégies) est un peu plus long à programmer ;
  - il comporte
    - un appels à chaque stratégie (avec le même historique),
    - un décompte des coups proposés, et
    - l'analyse des résultats pour choisir le plus courant.
  - Pour le décompte et l'analyse des résultats, cela ressemble aux stratégies qui cherchent dans l'historique le coup le plus fréquent pour l'humain et cherche à le contrer ;
  - mais la première partie (appel des stratégies) est différente.
- Globalement, comparés aux compte-rendu fournis pour le mini-projet, les résultats à l'exercice ne sont pas à la hauteur des travaux rendus. De nombreux compte-rendu ont fait des stratégies statistiques et devraient pouvoir les refaire le jour de l'examen.
  Rappel : le projet est proposé pour prolonger la partie Prolog, acquérir un peu de pratique et quelques habitudes de programmation, permettre de préparer le partiel de mi-semestre et

l'examen terminal de fin d'année, il faut le faire sérieusement et ne pas le déléguer à son binôme, humain ou artificiel.

2 Exercice 2, Indexe²

En ProLog :

- version simple (avec calcul de longueur en début)
  on voudrait pouvoir écrire (et sur le papier, ou au tableau, on peut c'est ok) :

```
indexe([],Deb,Fin,[]).
indexe([E|L],Deb,Fin,[[Deb,E,Fin]|R]):-
 indexe(L,Deb+1,Fin-1,R).

longueur([],0).
longueur([_E|L],N+1):-
  longueur(L,N).
```

mais pour diverses raisons, en machine, sauf à écrire un petit transpiler, il faut plutôt écrire :

[1]:
```
%%writefile prog.pl

indexe([],_,_,[]).
indexe([E|L],IndiceCroissant,IndiceDecroissant,
            [[IndiceCroissant,E,IndiceDecroissant]|R]):-
 indexe(L,IndiceCroissant_plus_1,IndiceDecroissant_moins_1,R),
 {IndiceCroissant_plus_1=IndiceCroissant+1},
 {IndiceDecroissant_moins_1=IndiceDecroissant-1}.

longueur([],-1).
longueur([_E|L],N_plus_1):-
  longueur(L,N), {N_plus_1=N+1}.

main :-
  writeln('L ?'), read(L),
  write('Pour L = '), write(L), write(', on obtient : '),
  longueur(L,X), indexe(L,0,X,R), write(R), writeln('.').

:- use_module(library(clpq)), main.
```

```
Overwriting prog.pl
```

ainsi, ça marche :

[2]:
```
!echo "[a,b,c,d,e]." | swipl -g halt -s prog.pl
```

```
L ?
Pour L = [a,b,c,d,e], on obtient : [[0,a,4],[1,b,3],[2,c,2],[3,d,1],[4,e,0]].
```

Rem. il y a des versions avec contraintes sans calcul de longueur explicite reposant sur le fait que

l'index croissant de début commence à 0 et que l'index décroissant à la fin termine avec 0 aussi. Par ex. (hors liste vide) :

[3]:
```
%%writefile prog.pl

indexe([E],Deb,0,[[Deb,E,0]]).
indexe([E|L],Deb,Fin,[[Deb,E,Fin]|R]):-
 indexe(L,DebL,FinL,R), {DebL=Deb+1,FinL=Fin-1}.

main :-
  writeln('L ?'), read(L),
  write('Pour L = '), write(L), write(', on obtient aussi : '),
  indexe(L,0,_,[[0,E,N]|R]), write([[0,E,N]|R]), writeln('.').

:- use_module(library(clpq)), main.
```

```
Overwriting prog.pl
```

[4]:
```
!echo "[a,b,c,d,e]." | swipl -g halt -s prog.pl
```

```
L ?
Pour L = [a,b,c,d,e], on obtient aussi :
[[0,a,4],[1,b,3],[2,c,2],[3,d,1],[4,e,0]].
```

En Erlang :

[5]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

indexe([],_,_)->[];
indexe([E|L],IndiceCroissant,IndiceDecroissant)->
  [[IndiceCroissant,E,IndiceDecroissant]|
    indexe(L,IndiceCroissant+1,IndiceDecroissant-1)].

longueur([]) -> -1;
longueur([_E|L]) -> longueur(L)+1.

main([]) ->
  compile:file(prog),
  L=[a,b,c,d,e],
  io:format("A partir de L = ~p on obtient ~p~n",[L,indexe(L,0,longueur(L))]).
```

```
Overwriting prog.erl
```

[6]:
```
!escript prog.erl
```

```
A partir de L = [a,b,c,d,e] on obtient [[0,a,4],
```

```
[1,b,3],
[2,c,2],
[3,d,1],
[4,e,0]]
```

Rem. pour éviter le calcul explicite de longueur au départ, on peut faire l'indexation en 2 temps (un temps pour l'indice croissant dans la descente récursive, un temps pour l'indice descendant dans la remontée récursive (prise à l'envers, cela donne un indice croissant aussi))
Par ex. :

[7]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

indexe([],_)->[];
indexe([E],Z)->[[Z,E,0]];
indexe([E|L],Z)->
  [[A,F,X]|R]=indexe(L,Z+1), %% rem. : A et Z son liés
  [[A-1,E,X+1],[A,F,X]|R].    %%a priori : ici A-1==Z, donc avant A==Z+1 (et␣
 ↪vice-versa)

main([]) ->
  compile:file(prog),
  L=[a,b,c,d,e],
  io:format("A partir de L = ~p on obtient aussi ~p~n",[L,indexe(L,0)]).
```

```
Overwriting prog.erl
```

[8]:
```
!escript prog.erl
```

```
A partir de L = [a,b,c,d,e] on obtient aussi [[0,a,4],
                                              [1,b,3],
                                              [2,c,2],
                                              [3,d,1],
                                              [4,e,0]]
```

Remarques globales :

- pour ProLog comme pour Erlang, l'exercice comportait une difficulté : pour un singleton [E] la bonne réponse est effectivement [[0,E,0]], mais quand la liste est plus longue, le dernier élément de liste [...,E] se traduit par [...,[LongueurDeLaListeAvant,E,0]] et non pas [...,[0,E,0]] ; aussi pour définir le cas de base des programmes il était préférable de pas utiliser ce cas singleton et utiliser la cas de base liste vide, ou un cas de base singleton avec l'information supplémentaire de la longueur de la liste avant.
- les preuves de correction, complétude, terminaison sont souvent verbeuses et parfois peu convaincantes (sans parler des preuves données alors que les programmes ne sont pas finis ou faux), il faut arriver à reste concis, précis :
  - correction pour le programme prolog : le cas de base est correct (évident) ; le cas de propagation de la récurrence avec les informations des indices croissants/décroissants est assez évident, sous réserve que ces informations sont correctes, c'est ce que l'on obtient

en regardant l'appel récursif et l'appel initial
ici les points importants, ce sont les indications des paramètres ajoutés (indices croissants/décroissants), de l'appel récursif et de l'appel initial ; ces points sont rarement les mêmes, d'un programme à l'autre, même dans les cas simples.

- complétude : le cas d'une liste vide ou non vide est prévu, sans restriction, tous les cas sont donc prévus (*argument usuel pour les cas simples*)
- terminaison : le seul appel récursif se déroule avec une liste de taille réduite de 1 (*argument usuel pour les cas simples*)
- attention, dans les copies, de nombreux programmes n'étaient pas "simples", l'argumentaire usuel utilisé ici pour la complétude et la terminaison ne serait pas adapté.

3 Exercice 3, triangle ou pyramide ?

[9]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

triangleMax(N,Max) when N*(N+1) div 2 > Max -> (N-1)*N div 2;
triangleMax(N,Max) -> triangleMax(N+1,Max).

pyramideMax(N,Max) when N*(N+1)*(2*N+1) div 6 > Max -> (N-1)*N*(2*N-1) div 6;
pyramideMax(N,Max) -> pyramideMax(N+1,Max).

max2(A,B) when A>B -> A;
max2(_A,B) -> B.

main([]) ->
  compile:file(prog),
  io:format("Triangle (max) = ~p~nPyramide (max) = ~p~ndonc Max = ~p~n",
                [triangleMax(0,1000000000), pyramideMax(0,1000000000),
                 max2(triangleMax(0,1000000000),pyramideMax(0,1000000000))]).
```

```
Overwriting prog.erl
```

[10]:
```
!time escript prog.erl
```

```
Triangle (max) = 999961560
Pyramide (max) = 998441521
donc Max = 999961560

real    0m0.281s
user    0m0.301s
sys     0m0.045s
```

Remarques :

- on peut aussi dire que c'est un nombre triangulaire et donner le "N" correspondant.

- il est plus prudent de ne pas le demander aux LLM, souvent (aujourd'hui), ils disent n'importe quoi, ex. :

```
Réponse finale :
  Plus grand nombre triangulaire à 9 chiffres :
  999 965 560

  Plus grand nombre pyramidal à 9 chiffres :
  999 999 000
```

Pour la version parallèle :

[11]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

coeurTriangle(N,M,Id) ->
  Id!triangleMax(N,M).

triangleMax(N,Max) when N*(N+1)/2 > Max -> (N-1)*N/2;
triangleMax(N,Max) -> triangleMax(N+1,Max).

coeurPyramide(N,M,Id) ->
  Id!pyramideMax(N,M).

pyramideMax(N,Max) when N*(N+1)*(2*N+1)/6 > Max -> (N-1)*N*(2*N-1)/6;
pyramideMax(N,Max) -> pyramideMax(N+1,Max).

max2(A,B) when A>B -> A;
max2(_A,B) -> B.

main([]) ->
  compile:file(prog),
  spawn(prog,coeurTriangle,[0,1000000000,self()]),
  spawn(prog,coeurPyramide,[0,1000000000,self()]),
  receive A -> receive B ->   io:format("Max = ~p~n",[max2(A,B)]) end end.
```

```
Overwriting prog.erl
```

[12]:
```
!time escript prog.erl
```

```
Max = 999961560.0

real    0m0.189s
user    0m0.224s
sys     0m0.032s
```

Remarques :

- le temps n'est pas divisé par 2, mais il y a tout de même un gain significatif (le lancement de

Erlang est dans la partie séquentielle incompressible)

- la version parallèle proposée conserve le code de la version séquentielle, on peut aussi faire un code plus court, mais qui risque d'être un peu moins lisible mélangeant calcul et parallélisation (à chacun de voir ce qu'il préfère, pour ma part, à-priori, je privilégierais la séparation du code de calcul et du code de parallélisation) :

[13]:
```
%%writefile prog.erl
-module(prog).
-compile([export_all,nowarn_export_all]).

coeurTriangle(N,Max,Id) when N*(N+1)/2 > Max -> Id!(N-1)*N/2;
coeurTriangle(N,Max,Id) -> coeurTriangle(N+1,Max,Id).

coeurPyramide(N,Max,Id) when N*(N+1)*(2*N+1)/6 > Max -> Id!(N-1)*N*(2*N-1)/6;
coeurPyramide(N,Max,Id) -> coeurPyramide(N+1,Max,Id).

main([]) ->
  compile:file(prog),
  spawn(prog,coeurTriangle,[0,1000000000,self()]),
  spawn(prog,coeurPyramide,[0,1000000000,self()]),
  receive A ->
    receive B when A>B -> io:format("Max = ~p~n",[A]);
             B         -> io:format("Max = ~p~n",[B]) end end.
```

```
Overwriting prog.erl
```

[14]:
```
!time escript prog.erl
```

```
Max = 999961560.0

real    0m0.190s
user    0m0.284s
sys     0m0.037s
```

4 Conclusions, histogramme des notes

Il y a longtemps que les notes n'avaient pas atteint le 20/20, félicitations à celui/celle qui a tout réussi (en juste un copie-double, bien rédigée et facile à lire) ; bravo aussi aux 15/20 et 16/20 ; et tout de même, mes encouragements à tous ceux qui sont au niveau de la moyenne ou presque (7-8/20), en espérant que les notes de CC ne sont pas trop mauvaises, j'espère que vous arriverez à avoir l'UE. Pour ceux qui sont encore un peu loin de l'objectif, il y a aussi la compensation globale et la seconde chance, mais il va falloir encore travailler pour arriver à programmer plus déclarativement.

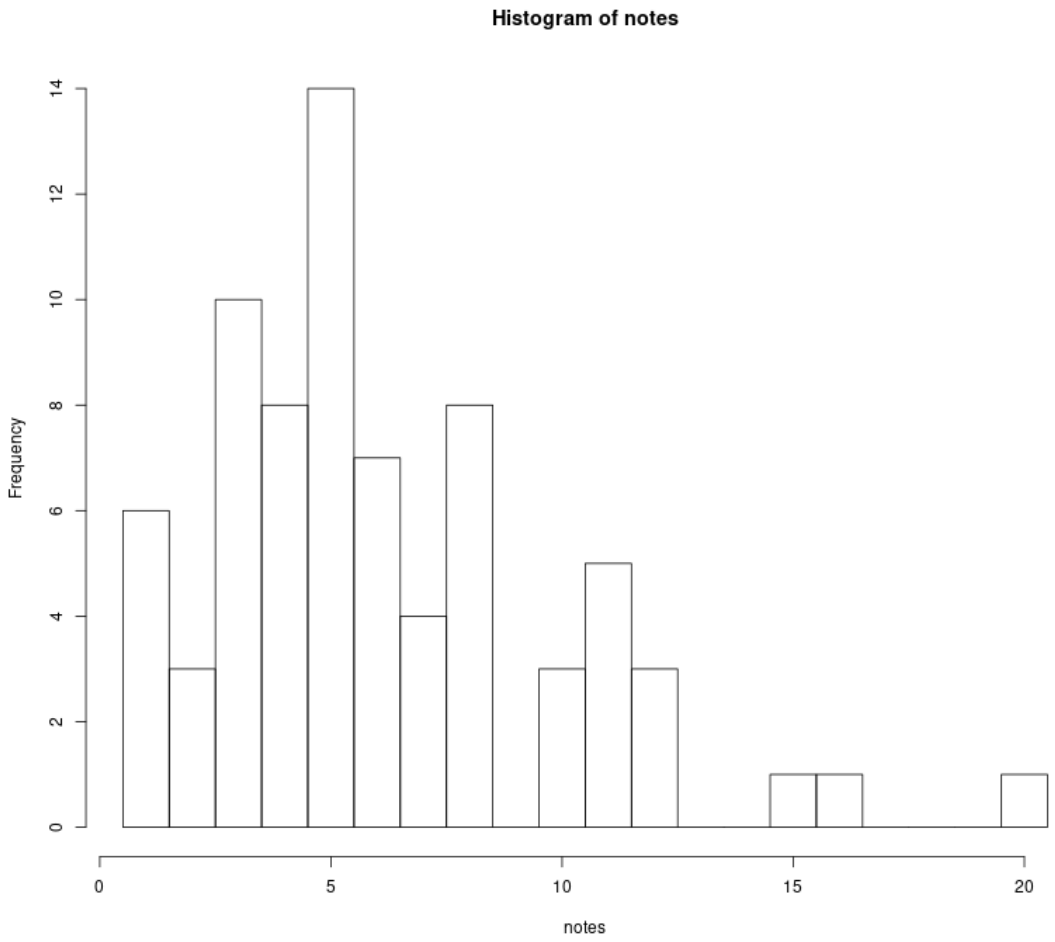
Pour la seconde chance, ou pour atteindre l'objectif (programmer plus déclarativement) :

- **raisonner par cas** (en général, c'est ok, juste souvent il y a trop de cas, n'ayez que le minimum de cas, les cas redondants ne servent à rien (sauf à compliquer la lecture, le contrôle et la validation))

- pensez **récursivité** : comment le problème globale se retrouve dans une partie seulement des données, (en général, la fin de la liste) et comment avec la solution pour une partie des données, “on” (re)construit la solution globale pour l’ensemble des données,
- vérifiez *vraiment* que votre programme est **correct**, **complet** et **termine**.

Et retournez sur Caseine (exercice, documents de cours et QCM), sur youtube (pour les vidéos sur prolog et erlang) et vous pouvez même demander aux LLM de vous aider à faire et à comprendre (mais ne leur demandez pas de faire à votre place, et ne faites pas comme si vous avez compris si ce n’est pas le cas ; insistez pour qu’ils vous aident vraiment à travailler, à comprendre et avancer par vous même ; pour que, la prochaine fois, ce soit vous qui fassiez tout tout/e seul/e)

Pour les notes (arrondies au demi-point supérieur) :



Mot final, comment savoir s’il faut demander aux LLM, à l’enseignant, à internet, aux autres étudiants, dans tous les cas :

- avant de poser une question, se demander si la question/réponse va vous aider à apprendre et si l’échange question/réponse est (éco-)responsable, raisonnable, éthique, etc.

Et après avoir écouter la réponse, se demander encore :

- si la réponse est *vraiment* correcte,
  et si vous la comprenez *vraiment* (si c'est le cas, vous devez pouvoir l'expliquer)

Enfin, si vous voulez que cet échange Question/Réponse servent vos apprentissages sur le long terme, essayer aussi de vous rappeler des solutions d'exercices anciens ou de re-chercher des exercices similaires sur le même thème, de temps en temps (en éloignant les temps d'exercice au fur et à mesure du temps qui passe ; avant d'oublier ce que vous avez appris, mais pas trop vite loin non plus, si c'est trop simple d'une fois sur l'autre, c'est une perte de temps, vous pouvez espacez plus : le premier rappel quelques jours après avoir appris et compris quelque chose de nouveau, ou, si les savoirs s'accumulent, entremêlez les rappels ; puis ensuite une semaine ou quinze jours puis un mois ou plus après)