NFP120 - Annales (tirée en grande partie de l'examen 2011)

1 Prolog

Ex. 1 — Écrire le prédicat `est_cinq(X)` qui est vrai si `X` est une liste de longueur exactement 5.

Solution (Ex. 1) —

```
est_cinq([_,_,_,_,_]).
```

Ex. 2 — Écrire le prédicat `est_mult_trois(X)` qui est vrai si `X` est une liste dont la longueur est un multiple de 3.

Solution (Ex. 2) —

```
est_mult_trois([]).
est_mult_trois([_,_,_|L]):-est_mult_trois(L).
```

Ex. 3 — Écrire le prédicat `get(L,N,X)` qui est vrai si `X` est l'élément numéro `N` de la liste `L` (le premier élément d'une liste est l'élément numéro 0). En particulier si `N` est un nombre négatif, le prédicat doit retourner `false` et ne pas boucler.

Solution (Ex. 3) —

```
get([X|_],0,X).
get([_|L],N,X):-N>0,N1 is N-1,get(L,N1,X).
```

Ex. 4 — Écrire le prédicat prolog sommeListe : `sommeListe(Liste1,Liste2, ListeSomme).` Telle que si Liste1 et Liste2 sont des listes d'entiers de même longueur, ListeSomme contienne la somme terme à terme de Liste1 et Liste2. Exemple :

```
?- sommeListe([3,2,3], [4,4,7], L).
L= [7, 6, 10].
```

Si Liste1 et Liste2 sont de longueur différentes, le prédicat doit échouer (c'est-à-dire que prolog ne trouve aucune solution).

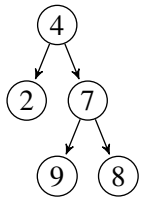
Solution (Ex. 4) —

```
sommeListe([],[],[]).
sommeListe([E1|L1],[E2|L2],[E3|L3]):- E3 is E1+E2, sommeListe(L1,L2,L3).
```

Ex. 5 — Pour cette question on utilise la représentation des arbres *binaires* vue en cours, avec deux symboles `t` et `feuille` définis comme suit :
–le terme `feuille(N)` représente l'arbre à un seul noeud contenant l'entier `N` ;
–le terme `t(N,X,Y)` représente l'arbre ayant l'entier `N` à la racine et les arbres `X` et `Y` comme sous-arbres ;
Par ailleurs l'arbre vide ne peut pas être représenté. Par exemple, le terme :

```
t(4,feuille(2),
    t(7,feuille(9),
        feuille(8))
 )
```

représente l'arbre suivant :



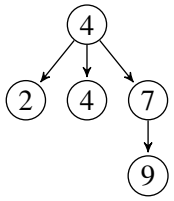
Écrire les prédicats `prof(X,N)` et `max(X,N)` définis comme suit : `prof(X,N)` est vrai si l'entier `N` est la profondeur de l'arbre `X` ; `max(X,N)` est vrai si `N` est le plus grand entier de l'arbre `X`.

Solution (Ex. 5) —

```
max(M,N,M):- M>=N.
max(M,N,N):- M<N.

prof(feuille(_),1).
prof(t(_,A,B),N):-prof(A,M),prof(B,P),max(M,P,N2),N is N2+1.


max3(M,N,P,M):- M>=N,M>=P.
max3(M,N,P,N):- N>M,N>=P.
max3(M,N,P,P):- P>M,P>N.

maxAB(feuille(N),N).
maxAB(t(N,A,B),M):-maxAB(A,MA),maxAB(B,MB),max3(N,MA,MB,M).
```

Ex. 6 — (dur) On adopte maintenant une représentation différente des arbres. On représente un arbre (pas nécessairement binaire) comme une liste comme suit :
–La liste vide `[]` représente l'arbre vide
–La liste `[N]` à un élément `N` entier représente la feuille contenant `N`
–La liste `[N|L]` représente le noeud contenant `N` et dont `L` est la liste des fils.
Par exemple :

`[4, [2], [4], [7,[9]] ]` correspond à l'arbre :



Écrire le prédicat `est_ABU(X)` qui est vrai si `X` est une arbre dont aucun noeud n'a plus de deux fils (c'est-à-dire que chaque noeud a zéro, un ou deux fils).

Solution (Ex. 6) —

```
est_ABU([]).
```

```
est_ABU([X]):-integer(X).
est_ABU([X,Y]):-integer(X), est_ABU(Y).
est_ABU([X,Y,Z]):-integer(X), est_ABU(Y), est_ABU(Z).
```

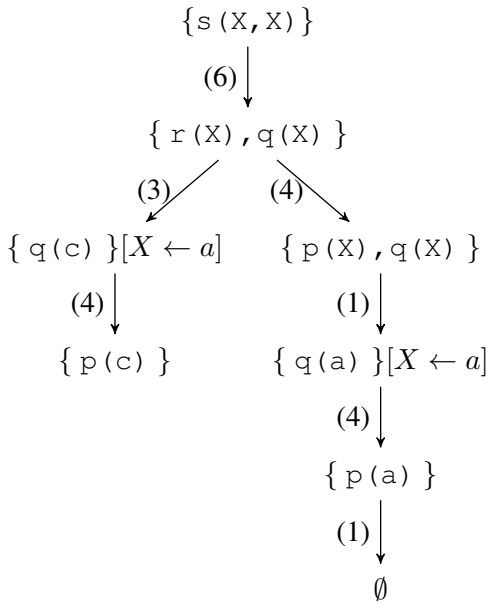
Ex. 7 — Donnez la sémantique en chaînage avant (ou par point fixe) du programme `datalog` suivant :

```
(1) p(a).          (2) q(b).          (3) r(c).          (4) q(X):-p(X).
        (6) r(X):-p(X),q(X).            (7) s(X,Y):-r(X),q(Y).
```

Solution (Ex. 7) — {`p(a)`, `~q(b)`, `~r(c)`, `~q(a)`, `~s(c,b)`, `~s(c,a)`, `~r(a)`, `~s(a,a)`, `~s(a,b)`}

Ex. 8 — 1,5pt Sur le même programme que la question précédente, donnez l'arbre de résolution de la question `?-~s(a,a)`. Indiquez sur chaque arête de l'arbre la règle utilisée et éventuellement la substitution utilisée.

Solution (Ex. 8) —



2 Tableaux sémantiques

Ex. 9 — En utilisant les règles de l'annexe A, montrez par les tableaux sémantiques le jugement suivant :

$$\forall x, y, z, \big(\ (s(x,y) \wedge s(y,z)) \rightarrow s(x,z)\ \big), \quad \forall x(\neg s(x,x)) \quad \vdash \quad \forall x, y, (s(x,y) \rightarrow \neg s(y,x))$$

Solution (Ex. 9) — Voir Annales 2005

3 Logique de Hoare

3.1 Invariant 1

Soit la boucle B suivante :

```
while(n>0) do
  a:=a+n;
  n:=n-1
done
```

Ex. 10 — Donnez un variant de cette boucle.

Solution (Ex. 10) — n

Ex. 11 — Donnez un invariant de cette boucle permettant de prouver le triplet de Hoare suivant (dans lequel n_0 et a_0 sont des constantes) : $\{\texttt{a} = a_0 \wedge \texttt{n} = n_0 \wedge n_0 \geq 0\} B \{\texttt{a} = a_0 + \sum_{i=1}^{n_0} i\}$

Solution (Ex. 11) — $\texttt{a}=\texttt{a}_0 + \sum_{i=n+1}^{n_0} i \wedge n \leq n_0 \wedge n \geq 0$

3.2 Invariant 2

Soit maintenant la boucle C suivante, qui inverse l'ordre des valeurs d'un tableau :

```
while(i>j) do
  tmp := t[i];
  t[i] := t[j];
  t[j] := tmp;
  i:=i+1;
  j:=j-1
done
```

Ex. 12 — Donnez un variant de cette boucle.

Solution (Ex. 12) — $j - i$

Ex. 13 — Donnez un triplet de Hoare de la forme `{P}` D `{Q}` exprimant la correction partielle de D. On utilisera la notation $\texttt{t}_0\texttt{[n]}$ pour désigner la $\texttt{n}^{\text{ème}}$ case du tableau `t` *avant la boucle*.

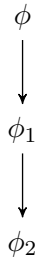
Solution (Ex. 13) — $\{\texttt{i}=i_0 \wedge \texttt{j}=j_0\}$ `D` $\{ \forall k, i_0 \leq k \leq j_0 \rightarrow \texttt{t[}k\texttt{]}=\texttt{t}_0\texttt{[}j_0\texttt{-(}k\texttt{-}i_0\texttt{)]} \}$

Ex. 14 — Donnez un invariant de la boucle suffisant pour en déduire la post-condition `Q` à la sortie de la boucle. Justifiez en expliquant comment déduire, à partir de l'invariant et de la condition de sortie de la boucle, que la post-condition est vérifiée après la boucle.

Solution (Ex. 14) — Il ne faut pas oublier que la symétrie `i`/`j` fait partie de l'invariant.

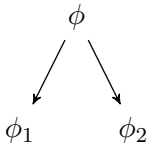
$\forall k, i_0 \leq k < \texttt{i} \rightarrow \texttt{t[}k\texttt{]}=\texttt{t}_0\texttt{[}j_0\texttt{-(}k\texttt{-}i_0\texttt{)]} \wedge \texttt{i}-i_0 = j_0-\texttt{j}$

A Annexe : Règle de la méthode des tableaux



ϕ	ϕ_1	ϕ_2
$\neg\neg\phi_1$	ϕ_1	
$\phi_1 \wedge \phi_2$	ϕ_1	ϕ_2
$\neg(\phi_1 \vee \phi_2)$	$\neg\phi_1$	$\neg\phi_2$
$\neg(\phi_1 \rightarrow \phi_2)$	ϕ_1	$\neg\phi_2$
$\phi_1 \leftrightarrow \phi_2$	$\phi_1 \rightarrow \phi_2$	$\phi_2 \rightarrow \phi_1$
$\exists x, (\phi_1(x))$	$\phi_1(a_i)$	
$\forall x, (\phi_1(x))$	$\phi_1(c)$	
$\neg\exists x, (\phi_1(x))$	$\neg\phi_1(c)$	
$\neg\forall x, (\phi_1(x))$	$\neg\phi_1(a_i)$	

FIGURE 1 – Les règles de type « et », qui ne créent pas d'embranchement. a_i désigne une nouvelle constante « fraîche » et c désigne une constante existante.



ϕ	ϕ_1	ϕ_2
$\phi_1 \vee \phi_2$	ϕ_1	ϕ_2
$\neg(\phi_1 \wedge \phi_2)$	$\neg\phi_1$	$\neg\phi_2$
$\phi_1 \rightarrow \phi_2$	$\neg\phi_1$	ϕ_2
$\neg(\phi_1 \leftrightarrow \phi_2)$	$\neg(\phi_1 \rightarrow \phi_2)$	$\neg(\phi_2 \rightarrow \phi_1)$

FIGURE 2 – Les règles de type "ou", qui créent un embranchement