Correction du devoir du Vendredi 9 Juin

Exercice 1

1. (a) Voici les primitives d'accès demandées :

```
let creer_pile () = {contenu = []} ;;

let etre_pile_vide p = match p.contenu with
    | [] -> true
    | _ -> false
;;

let empiler e p = p.contenu <- e::p.contenu ;;

let depiler p = match p.contenu with
    | [] -> failwith "pile vide"
    | t::q -> p.contenu <- q; t
;;
```

(b) Voici la fonction `insere_pile` :

```
let rec insere_pile x p =
  if etre_pile_vide p then
    {contenu = [x]}
  else
    begin
      let t = depiler p in
        if x <= t then
          empiler x (empiler t p)
        else
          empiler t (insere_pile x p)
    end
;;
```

(c) Voici la fonction `tri_insertion_pile p` :

```
let rec tri_insertion_pile p =
  if etre_pile_vide p then
    p
  else
    insere_pile depiler p (tri_insertion_pile p)
;;
```

2. (a) Voici les instructions pour définir l'arbre de l'énoncé :

```
let a = N(2, N(1, Vide, Vide), N(4, N(7, Vide, Vide), Vide)) ;;
```

(b) Voici la fonction `noeuds` :

```
let rec noeuds a = match a with
  | Vide -> 0
  | N(_, Vide, Vide) -> 0
  | N(_, g, d) -> 1 + noeuds g + noeuds d
;;
```

(c) Voici la fonction `hauteur` :

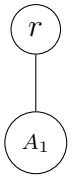
```
let rec hauteur a = match a with
  | Vide -> -1
  | N(_, g, d) -> 1 + max (hauteur g) (hauteur d)
;;
```

(d) Par induction structurelle à partir de la définition récursive d'arbre.

Cas de base : Soit A un arbre binaire à 0 noeud. A possède est de hauteur 0 et on a bien la relation demandé.

Etape d'induction : On considère un arbre binaire A à n noeuds. Notons, pour tout arbre B, $N_n(B)$ le nombre de noeuds de B et $h(B)$ sa hauteur. On a deux cas :

— Soit A est de la forme



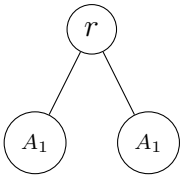
où r est la racine d'arité 1 et A_1 est un arbre binaire non vide vérifiant la propriété. Alors, en utilisant l'hypothèse d'induction structurelle :

$$h(A) = 1 + h(A_1) \leqslant 1 + N_n(A_1) = N_n(A)$$

et

$$\begin{aligned} N_n(A) &= N_n(A_1) + 1 \leqslant 2^{h(A_1)} - 1 + 1 = 2^{h(A_1)} \\ &\leqslant 2^{h(A_1)} + \underbrace{2^{h(A_1)} - 1}_{0\leqslant} = 2^{h(A_1)+1} - 1 = 2^{h(A)} - 1. \end{aligned}$$

— Soit A est de la forme



où r est la racine d'arité 2 et A_1, A_2 sont des arbres binaires non vides vérifiant la propriété. Supposons par exemple $h(A_1) \geqslant h(A_2)$. Alors, en utilisant l'hypothèse d'induction structurelle :

$$h(A) = 1 + h(A_1) \leqslant 1 + N_n(A_1) \leqslant 1 + N_n(A_1) + N_n(A_2) = N_n(A)$$

et

$$\begin{aligned} N_n(A) &= 1 + N_n(A_1) + N_n(A_2) \leqslant 1 + 2^{h(A_1)} - 1 + 2^{h(A_2)} - 1 \\ &\leqslant 2^{h(A_1)} + 2^{h(A_1)} - 1 = 2^{h(A)} - 1. \end{aligned}$$

Par induction structurelle, le résultat est vrai.

(e) Pour parcourir un arbre en profondeur en mode préfixe, on procède récursivement en visitant entièrement les sous-arbres de gauche à droite et on traite les sommets visités à la première visite.

(f) Voici la fonction `prefixe` demandée :

```
let rec prefixe a = match a with
  | N(r, Vide, Vide) -> [r]
  | N(r, g, d) -> (r :: prefixe g) @ (prefixe d)
;;
```

Exercice 2

Pour la commodité d'écriture, on utilisera parfois les notations 0 et 1 pour `Faux` et `Vrai`.

Q.1 a. $x_1 \wedge (x_0 \vee \neg x_0) \wedge \neg x_1$ n'est pas satisfiable (vaut 0 que x_1 soit égal à 0 ou 1).
b. $(x_0 \vee \neg x_1) \wedge (\neg x_0 \vee x_2) \wedge (x_1 \vee \neg x_2)$ est satisfiable (donner à tous les x_i la valeur 1).
c. $x_0 \wedge \neg(x_0 \wedge \neg(x_1 \wedge (x_1 \wedge \neg x_2)))$ est satisfiable (prendre $x_0 = 1$ et $x_1 = 0$).
d. $(x_0 \vee x_1) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_1)$ n'est pas satisfiable (les deux premières disjonctions imposent $x_1 = 1$ et les deux dernières $x_1 = 0$).

Q.2 Une première fonction (de type `clause → int`) donne l'indice maximum d'une variable présente dans une clause. Le cas d'une clause vide est particulier. On pourrait l'exclure en renvoyant une erreur ou exception. On peut aussi convenir que l'indice maximal d'une clause vide est l'entier minimum en OCaml. On vérifie que cette convention est cohérente avec le cas d'une clause de longueur 1.

```
let rec var_max_clause c = match c with
  | [] -> min_int
  | (V x)::q -> max x (var_max_clause q)
  | (NV x)::q -> max x (var_max_clause q)
;;
```

Le principe est le même pour la fonction demandée en itérant sur les clauses.

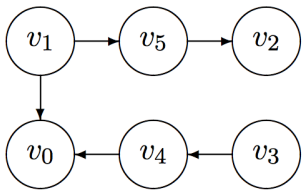
```
let rec var_max f = match f with
  | [] -> min_int
  | c::q -> max (var_max_clause c) (var_max q)
;;
```

Q.3 On commence par initialiser un tableau assez grand de triléens ; sa taille dépend bien sur du numéro maximal de variable. On itère ensuite sur les clauses supposées de taille 1 avec une fonction `aux_un_sat : clause → booléen`. Cette fonction indique si la formule (supposée être une $1-FNC$) est satisfiable ET met à jour le tableau des triléens.

```
let un_sat f =
  let sigma=Array.make (1+(var_max f)) Indetermine in
  let rec aux_un_sat f = match f with
    | [] -> true
    | [V x]::q -> if sigma.(x)=Faux then false
```

```
                    else if sigma.(x)=Vrai then aux_un_sat q
                    else begin
                       sigma.(x) <- Vrai ;
                       aux_un_sat q
                       end
      | [NV x]::q -> if sigma.(x)=Vrai then false
                    else if sigma.(x)=Faux then aux_un_sat q
                    else begin
                       sigma.(x) <- Faux ;
                       aux_un_sat q
                       end
in aux_un_sat f;;
```

Q.4 La clause $(x_2 \vee \neg x_2)$ disparait. Comme on a deux clauses de longueur 2 et une de longueur 1, on obtient 5 arêtes :



Q.5 Après avoir défini un graphe sans arête de bonne taille, on écrit une fonction auxiliaire `parcourt` qui prendre en argument une clause de longueur $\leqslant 2$ (en éliminant les cas dégénérés évoqués par l'énoncé) et qui ajoute les arêtes associées au graphe. Il suffit de la faire agir sur chaque clause et de renvoyer le graphe.

```
let deux_sat_vers_graphe f=
  let g = Array.make (2*(1+(var_max f))) [] in
  let ajout_clause c = match c with
    | [V i] -> g.(2*i+1) <- (2*i)::g.(2*i+1)
    | [NV i] -> g.(2*i) <- (2*i+1)::g.(2*i)
    | [V i;V j] -> g.(2*i+1) <- (2*j)::g.(2*i+1);
                   g.(2*j+1) <- (2*i)::g.(2*j+1)
    | [V i;NV j] -> g.(2*i+1) <- (2*j+1)::g.(2*i+1);
                    g.(2*j) <- (2*i)::g.(2*j)
    | [NV i;V j] -> g.(2*i) <- (2*j)::g.(2*i);
                    g.(2*j+1) <- (2*i+1)::g.(2*j+1)
    | [NV i;NV j] -> g.(2*i) <- (2*j+1)::g.(2*i);
                     g.(2*j) <- (2*i+1)::g.(2*j)
  in
  List.iter ajout_clause f;
  g;;

let f_ex = [[V 1; V 2]; [V 0]; [NV 2; V 0]];;

let g= deux_sat_vers_graphe f_ex;;
```

Q.6 Les sommets de G correspondent à des littéraux. Dans la suite, on utilise sans le préciser cette identification.

Soient v_i et v_j deux sommets situés dans une même composante. Il existe donc un chemin $\ell_0, \ldots, \ell_p$ dans G avec $\ell_0 = v_i$ et $\ell_p = v_j$, les ℓ_k étant des littéraux. Comme le graphe possède les arêtes (ℓ_k, ℓ_{k+1}), la formule contient l'une des clauses $\neg\ell_k \vee \ell_{k+1}$ ou $\neg\ell_{k+1} \vee \ell_k$. L'une de ces clauses est ainsi vérifiée par σ ce qui indique (avec les remarques de l'énoncé) que $\ell_k \Rightarrow \ell_{k+1}$ l'est aussi pour tout k. Ceci étant vrai pour tout k, il en résulte que $\ell_0 \Rightarrow \ell_p$ est aussi vérifiée par σ. Comme il existe aussi un chemin de ℓ_p vers ℓ_0, on en déduit que $\ell_p \Rightarrow \ell_0$ est aussi vérifiée par σ. Finalement, ℓ_p et ℓ_0 ont même valeur de vérité.

Les variables associées à v_i et v_j ont donc même valeur de vérité si $j-i$ est pair (car les littéraux ℓ_0 et ℓ_p ont alors même signe) et des valeurs de vérité opposées sinon. C'est ce qu'il convenait de montrer.

Q.7 Deux littéraux dans la même composante ayant même valeur de vérité par une valuation qui satisfait f, une même composante du graphe ne peut contenir un littéral et son opposé dans le cas satisfiable.

Q.8 Pour tester si la formule 2-SAT f est satisfiable, on forme (en temps linéaire en la taille de f) le graphe associé puis (en temps linéaire en la taille du graphe et donc aussi linéaire en la taille de f) la liste des composantes fortement connexes de g. On doit alors vérifier qu'aucune de ces composantes ne contient deux éléments du type $2i$ et $2i+1$. Pour conserver la complexité linéaire, il faut faire cela efficacement, en n'effectuant qu'un seul passage dans chaque liste représentant une composante. Ces listes n'étant pas triées, cela ne me semble pas immédiat. Je propose alors de commencer par introduire un tableau t de taille égale au nombre de sommets. Par un unique parcours de chaque liste, on peut remplir le tableau t en sorte que $t.(i)$ contienne le numéro d'un sommet référence de sa composante (par exemple le premier élément de la liste obtenue).

La fonction `remplir` prend en argument une liste correspondant à une composante et remplit le tableau t comme indiqué. Il reste à l'utiliser sur toutes les composantes. La boucle finale effectue la vérification sur t expliquée plus haut.

```
let deux_sat f =
  let g=deux_sat_vers_graphe f in
  let cf=cfc g in
  let n=Array.length g in
  let t=Array.make n 0 in
  let remplir_t l = List.iter (fun j -> t.(j)<-hd l) l in
  List.iter remplir_t cf;
  let i=ref 0 in
  while !i<n/2 && t.(2* !i)<>t.(2* !i+1) do i:= !i+1 done;
  !i=n/2;;
```

Q.9 Les fonctions sont élémentaires. On essaye de limiter le nombre des tests.

```
let et a b =
   if b=Faux||a=Faux then Faux
   else if a=Vrai&&b=Vrai then Vrai
   else Indetermine;;

let ou a b =
   if a=Vrai||b=Vrai then Vrai
```

```
    else if a=Faux&&b=Faux then Faux
    else Indetermine;;

let non = function
  | Vrai -> Faux
  | Faux -> Vrai
  | Indetermine -> Indetermine ;;
```

Q.10 Une première fonction `evalclause` effectue l'évaluation d'une clause (la fonction est locale et a donc accès au tableau `t` argument de `eval`). On peut conclure quand on trouve un littéral valant `Vrai` et continuer l'exploration sinon. On vérifie la cohérence du choix de valeur pour une clause vide en regardant ce qui se passe pour une clause de longueur 1. Il s'agit ensuite, dans la fonction `evalformule`, d'évaluer les clauses successives. On peut s'arrêter quand l'une vaut `Faux`. Sinon, tout dépend de la valeur de la clause et de celle du reste de la formule. On vérifie de même la cohérence du choix de valeur pour une formule vide.

```
let eval f t =
  let rec evalclause c = match c with
    | [] -> Faux
    | (V i) :: q -> if t.(i)=Vrai then Vrai
                    else let tt=evalclause q in ou t.(i) tt
    | (NV i) :: q -> if t.(i)=Faux then Vrai
                     else let tt=evalclause q in ou tt (non t.(i))
  in
    let rec evalformule f = match f with
       | [] -> Vrai
       | c :: q -> let tt = evalclause c in if tt=Faux then Faux
                                            else et tt (evalformule q)
    in evalformule f;;

eval f_ex [|Vrai; Faux; Indetermine|];;

eval f_ex [|Vrai; Faux; Vrai|];;
```

Q.11 Notre fonction principale va gérer une fonction auxiliaire `explore` prenant en argument un entier $k \in [|0, n|]$ correspondant au nombre de variables ayant reçu une valeur booléenne. Cette fonction doit renvoyer un booléen indiquant si cette valuation triléenne peut être complétée en une valuation booléenne qui satisfait notre formule. Elle est donc de type `int` $\rightarrow$ `bool`. Son schéma est le suivant

```
On calcule l'evaluation trileenne de la formule.
Si elle vaut Vrai alors, on renvoie True
Si elle vaut Faux alors, on renvoie False
Sinon
     on pose t.(k)=Vrai et on fait un appel recursif avec k+1
     Si l'appel renvoie True, on renvoie True
     Sinon
         on pose t.(k)=Faux et on fait un appel recursif avec k+1
         Si l'appel renvoie True, on renvoie True
```

```
        Sinon on pose t.(k)=Indetermine (remontée dans l'arbre)
        et on renvoie False
```

La fonction `var_max` permet de connaitre la taille du tableau à utiliser. La suite est la traduction OCaml de l'algorithme de l'énoncé.

```
let k_sat f =
  let n=var_max f in
  let t=Array.make (n+1) Indetermine in
  let rec explore k =
    let b=eval f t in
    if b=Vrai then true
    else if b=Faux then false
    else begin
       t.(k) <- Vrai ;
       if explore (k+1) then true
       else begin
          t.(k) <- Faux ;
          if explore (k+1) then true
          else begin
             t.(k) <- Indetermine ;
             false
           end
         end
       end
   in explore 0;;
```

Q.12 a. $(x_1 \vee \neg x_0) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_4 \vee x_2)$.

b. $(x_0 \vee x_2 \vee x_4) \wedge (x_0 \vee x_2 \vee x_5) \wedge (x_0 \vee x_3 \vee x_4) \wedge (x_0 \vee x_3 \vee x_5) \wedge$
$(x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_5) \wedge (x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_3 \vee x_5)$.

Q.13 La première étape conserve la satisfiabilité puisqu'on n'utilise que les lois de De Morgan. La seconde étape travaille avec des formules de l'ensemble $\mathcal{F}'$ défini récursivement à partir des littéraux (cas de base) en utilisant uniquement les connecteurs $\vee$ et $\wedge$. Il s'agit de montrer que pour toute formule f de $\mathcal{F}'$, l'étape b conserve la satisfiabilité (c'est à dire que f et f' sont équisatisfiables). Plus précisément, on montre que pour $f \in \mathcal{F}'$,

- si f est satisfaite par une valuation σ alors il existe une valuation σ' satisfaisant f' et telle que σ et σ' ont même action sur les variables présentes dans f ;
- si f' est satisfaite par une valuation σ' alors σ' satisfait aussi f.

On procède par induction structurelle.

- Le résultat est immédiat dans le cas de base car on ne modifie alors pas la formule.
- Soient ϕ et ψ deux éléments de $\mathcal{F}'$ vérifiant l'hypothèse.
  Si $f = \phi \wedge \psi$ alors $f' = \phi' \wedge \psi'$.
  - Supposons f satisfiable par σ ; ϕ et ψ le sont alors aussi. Par hypothèse d'induction, on peut alors trouver des valuations σ_1 et σ_2 satisfaisant ϕ' et ψ'. De plus, σ_1 et σ_2 ont la même action que σ sur les variables de f. Les variables supplémentaires dans ϕ' et dans ψ' ne sont pas les mêmes (ce n'est pas clair dans l'énoncé de la méthode mais c'est le cas si on en juge par

l'exemple donné dans l'énoncé). On peut donc poser σ' qui agit comme σ sur les variables de f, comme σ_1 sur les variables supplémentaires de ϕ' et comme σ_2 sur les variables supplémentaires de ψ'. σ' satisfait f' et a la même action que σ sur les variables de f.

- Supposons que f' est satisfiable. Il existe alors une valuation σ' qui satisfait ϕ' et ψ'. Par hypothèse d'induction, elle satisfait alors ϕ et ψ et donc aussi f.

Si $f = \phi \vee \psi$, on note x la variable supplémentaire introduite et qui n'est présente ni dans ϕ' ni dans ψ'.

- Si f est satisfaite par σ alors σ satisfait ϕ ou ψ. Les cas étant symétriques, supposons qu'elle satisfait ϕ. Par hypothèse d'induction, il existe σ' qui satisfait ϕ'.
  En considérant σ' qui satisfait ϕ' et en imposant $\sigma'(x) = Faux$ (ce qui est possible car pour satisfaire ϕ', la valeur de x n'importe pas), on trouve une valuation qui satisfait f'.
- Si f' est satisfaite par une valuation σ'. Si $\sigma'(x) = Faux$, alors σ' satisfait ϕ' et sinon elle satisfait ψ'. Dans le premier cas, par hypothèse d'induction, elle satisfait ϕ et donc f. Dans le second, par hypothèse d'induction, elle satisfait ψ et donc aussi f.

Q.14 C'est un processus récurrent simple. Il faut juste être attentif à ne pas oublier de cas.

```
let rec negs_en_bas f = match f with
  | Var i -> Var i
  | Et(a,b) -> Et(negs_en_bas a,negs_en_bas b)
  | Ou(a,b) -> Ou(negs_en_bas a,negs_en_bas b)
  | Non(Var i) -> Non (Var i)
  | Non(Et(a,b)) -> Ou(negs_en_bas(Non a),negs_en_bas(Non b))
  | Non(Ou(a,b)) -> Et(negs_en_bas(Non a),negs_en_bas(Non b))
  | Non(Non a) -> negs_en_bas a;;
```

Q.15 On applique l'algorithme assez naïvement sans essayer de réduire le nombre de concaténations. On gère une référence k indiquant le numéro de la dernière variable utilisée (afin de savoir quel numéro donner à une nouvelle variable à introduire).

```
let formule_vers_fnc f=
  let n=var_max_formule f in
  let k=ref n in
  let rec construit_fnc f = match f with
    | Var i -> [[V i]]
    | Non(Var i) -> [[NV i]]
    | Et(a,b) -> (construit_fnc a)@(construit_fnc b)
    | Ou(a,b) -> incr k;
                 let v= !k in
                 let fcb=map (fun l -> (NV v)::l) (construit_fnc b) in
                 let fca=map (fun l -> (V v)::l) (construit_fnc a) in
                 fca@fcb
  in construit_fnc f;;
```