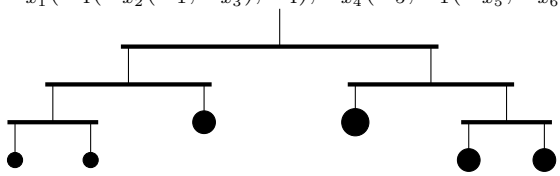
Correction Partiel 2021

Valeran MAYTIE

Exercice 1 – Récurrence

1. $B_{x_1}(B_4(B_{x_2}(O_1, O_{x_3}), O_4), B_{x_4}(O_5, B_1(O_{x_5}, O_{x_6})))$



$x_1 \in \mathbb{R}$
$x_2 = 2$
$x_3 = 1$
$x_4 = 2$
$x_5 = 2$
$x_6 = 2$

2. **Cas de base** $m = O_p$: On a bien $o(m) = 1$ et $b(m) = 0$ donc $b(m) + 1 = o(m)$
Cas récursif $m = B_p(m_1, m_2)$: On suppose la propriété vrai pour m_1 et m_2 (H.P)

$$\begin{aligned} o(B_p(m_1, m_2)) &= o(m_1) + o(m_2) && \text{Par construction de } o(m) \\ &= b(m_1) + 1 + b(m_2) + 1 && \text{Par H.P} \\ &= b(m_1) + b(m_2) + 1 + 1 \\ &= b(B_p(m_1, m_2)) + 1 && \text{Par construction de } b(m) \end{aligned}$$

3. $$masse(m) = \begin{cases} masse(O_p) = p \\ masse(B_p(m_1, m_2)) = p + masse(m_1) + masse(m_2) \end{cases}$$

4. On prend $y \in mathbbN$
Cas de base $m = O_p$: On a bien $masse(alourdir(O_p, y)) = masse(O_{p+y}) = p + y = masse(O_p) + y$
Cas récursif $m = B_p(m_1, m_2)$: On suppose la propriété vrai pour m_1 et m_2 (H.P)

$$\begin{aligned} masse(allourdir(B_p(m_1, m_2)), y) &= masse(B_{p+y/3}(alourdir(m_1, y/3), alourdir(m_2, y/3))) \\ &= p + y/3 + masse(alourdir(m_1, y/3)) + masse(alourdir(m_2, y/3)) \\ &= p + y/3 + masse(m_1) + y/3 + masse(m_2) + y/3 \\ &= p + masse(m_1) + masse(m_2) + y \\ &= masse(B_p(m_1, m_2)) + y \end{aligned}$$

5. Definition Ocaml de $masse(m)$ et $stable(m)$:

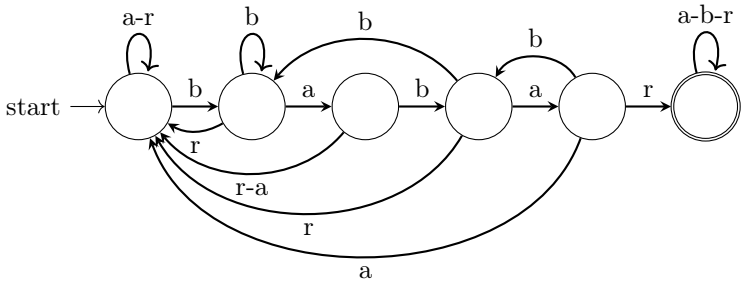
```
let rec masse = function
    | O (x) ->  x
    | B (x, m1, m2) -> x + masse m1 + masse m2

let stable = function
    | O (x) -> true
    | B(x, m1, m2) -> masse m1 = masse m2 && stable m1 && stable m2
```

Exercice 2 – Automate

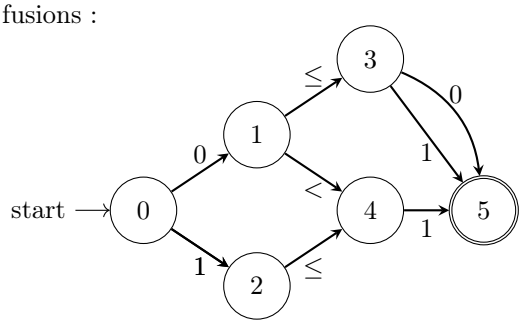
1. `(a|b|r)*babar(a|b|r)*`

2. Automate déterministe et complet reconaissant L :

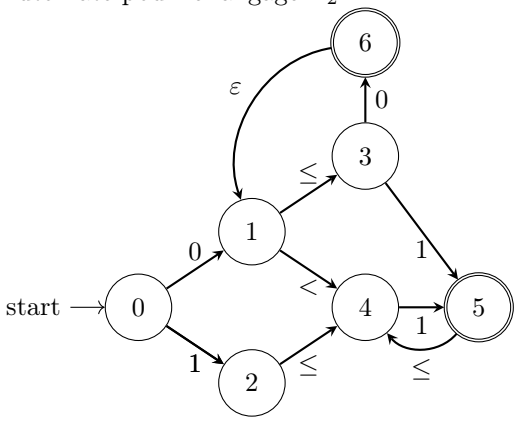


Exercice 3 – Langages reconnaissables

1. $((0((\leq (0|1))| < 1))|(1 \leq 1))$
2. fusions :



3. Automate pour le langage L_2 :



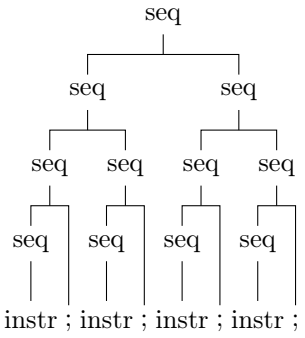
4. On suppose que L_3 est reconnaissable. On a $N \in \mathbb{N}$ telle que :
   $1^N \leq 1^N \in L_3$ on décompose ce mot en $m_0 m m_1$

$$\begin{aligned} m_0 &= \varepsilon \\ m &= 1^N \\ m_4 &= \leq 1^N \end{aligned}$$

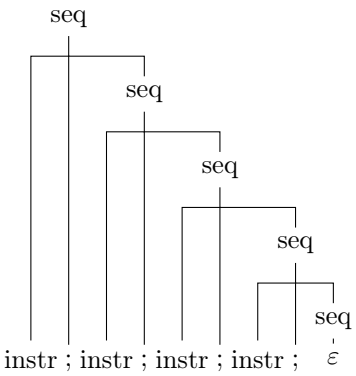
   On décompose $m = 1^N = 1^{n_1}1^{n_2}1^{n_3} = m_1 m_2 m_3$ avec $n_2 \neq 0$ et $n_1 + n_2 + n_3 = N$.
   Par le lemme de l'étoile on a $m_0 m_1 {m_2}^2 m_3 m_4 \in L_3$ donc $1^{n_1+2n_2+n_3} \leq 1^N \in$ or $n_1 + 2n_2 + n_3 = N + n_2 > N$
   On a donc une contradiction donc L_3 n'est pas reconnaissable.

Exercice 4 – Grammaires

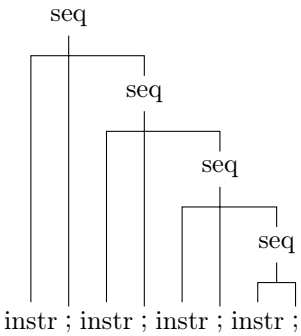
1. G_b :



G_c :



G_e :



2. G_a et G_b sont ambigues.
3. G_b et G_e décrivent le même langage
4. — G_a : `instr`
   — G_c : `instr;`
   — G_d : `;instr`

Exercice 5 – Interprétation

1. — `let x = 1 + 2 in e` : (e la suite) : $\rho = \{(\text{“}x'', Add(1,2))\}$
   — `let y = 1 + 4 in e` : (e la suite) : $\rho = \{(\text{“}x'', Add(1,2)); (\text{“}y'', Add(1,4))\}$
   — `eval(Add(Var("x")), eval(Var("x")))` $\rho = \{(\text{“}x'', Add(1,2)); (\text{“}y'', Add(1,4))\}$
   — `eval(Var("x"))` $\rho = \{(\text{“}x'', Cst(3)); (\text{“}y'', Add(1,4))\}$
   — `eval(Var("x"))` $2^{\text{ème}}$ évaluation $\rho = \{(\text{“}x'', Cst(3)); (\text{“}y'', Add(1,4))\}$
   — $\rho = \{\}$

   L'addition 1+2 est réalisée 1 fois
   L'addition 1+4 n'est pas réalisée
   C'est la comportement de la stratégie de *l'appel par nécessité.*
2. Le **replace** sert à remplacer l'encienne valeur de x par la nouvelle valeur évaluée ce qui permet de ne pas refaire de calcule à la deuxième évalutation de la variable. Si on l'enlève la variable sera juste réévaluée à chaque fois qu'on l'utilise.

3. Le **remove** sert à suprimer x de la table ce qui empêche d'y accéder quand la variable n'est plus à notre portée. Si on enlève cette expression les variables ne se supprimeront plus de la hastable. Mais comme la variable est évaluée quand elle est utilisée pour la première fois il n'y aura pas de problème. La table sera juste remplie plus rien.
   Exemple :

```
let x =
    let x = 3 in
    x + 2
in
x + x
```

   donne bien 10

4. définition de `eval: env -> expr -> epxr * env`

```
let rec eval env = function
    | Cst n -> (n, env)
    | Add(e1, e2) ->
        let v1, env1 = eval env e1 in
        let v2, env2 = eval env1 e2 in
        (v1 + v2, env2)
    | Var x ->
        let e = find env in
        let v, env' = eval env e in
        (v, env')
    | Let(x, e1, e2) ->
        let env' = add x e1 in
        let v, env' = eval e2, env' in
        (v, env)
```