LIFAP5 – Programmation fonctionnelle pour le WEB
TD1 – introduction au λ-calcul et à JavaScript

Licence informatique UCBL – Printemps 2021–2022

Exercice 1 : Typage et évaluation en λ-calcul

1. Pour chacune des λ-expressions suivantes, donner son type en supposant que le seul type primitif est `number` puis l'évaluer *en explicitant chaque β-réduction*. Si l'expression n'est pas typable, expliquer pourquoi. On supposera que l'addition notée $+$ a pour type `number` $\rightarrow$ `number` $\rightarrow$ `number` et se réduit par calcul arithmétique usuel, de même que pour la multiplication notée $*$. Par exemple, $(\lambda x.x + 3)\ 2$ a pour type `number` et peut se réduire en $(\lambda x.x + 3)\ 2 \overset{x}{\rightsquigarrow} 2 + 3 \rightsquigarrow 5$
   1. $(\lambda x.\lambda y.(x + y))\ 3\ 4$
   2. $(\lambda x.\lambda y.(x\ y))\ 3\ 4$
   3. $\lambda x.(x\ x)$
   4. $(\lambda x.\lambda y.(y\ (x + 2)))\ 5\ (\lambda z.(z * 3))$
   5. $(\lambda x.((\lambda y.(x + y))\ x))\ 5$
   6. $(\lambda x.((x\ (\lambda y.(y + 2))) + (x\ (\lambda z.(z * 3)))))\quad ((\lambda u.\lambda w.(w\ u))\ 5)$
   7. $(\lambda x.\lambda y.(x\ y))(\lambda z.z)$
   8. $\lambda x.(x + 3)(\lambda y.y)$
   9. $(\lambda x.\lambda y.x)$
   10. $(\lambda x.\lambda y.x)\ 3$
   11. $(\lambda x.\lambda y.(y\ (3 + x)))\ 4\ (\lambda z.(z * 2))$
   12. $(\lambda x.((\lambda y.(y * x))\ x))\ 5$
2. Donner deux séquences de réductions *différentes* de l'expression $(\lambda x.x + 5)((\lambda z.z)\ 2)$

Exercice 2 : Du λ-calcul au JavaScript et vice-versa

1. Réécrire les expressions suivantes du λ-calcul en JavaScript en utilisant la notation `=>` :
   — $\lambda f.\lambda x.f(f\ x)$
   — $\lambda x.\lambda y.(x\ y)\ y$
   — $\mathbf{I} = \lambda x.x$
   — $\mathbf{K} = \lambda x.\lambda y.x$
   — $\mathbf{S} = \lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$
2. Montrer que $\mathbf{SKK}x$ et $\mathbf{I}x$ se β-réduisent en le même terme. Conclure sur $\mathbf{I}$.
3. Réécrire les fonctions JavaScript suivantes en λ-calcul :
   — `let fn1 = f => g => x => f(g(x))`
   — `let fn2 = f => g => x => g(x)+ f(x)`
4. Soient les fonctions JavaScript suivantes `let f0 = x => 2 + x` et `let f1 = x => 2 * x`. Calculer `fn1(f0)(f1)(3)`.

5. Réécrire en JavaScript une version β-réduite (plus simple) de `fn1(f0)(f1)` avec les fonctions `f0` et `f1` de la question précédente.
6. Soit la fonction `let fn3 = x => x(x)`. Donner le résultat ou le message d'erreur qu'affiche la console JavaScript lors de l'exécution des expressions suivantes :
   — `fn3(3)`
   — `fn3(x => x)`
   — `fn3(x => x)(3)`
   — `fn3(fn3(x => x))(3)`
7. Considérons la fonction JavaScript `let w = x => x(x)`. L'exécution de `w(x => x +1)` produit le résultat `"x => x +11"`. Expliquez.

Exercice 3 : Codage dans le λ-calcul

Dans cet exercice on va s'intéresser au codage des booléens en λ-calcul pur appelé *booléens de Church*. Dans ce codage, les valeurs de vérités sont *des fonctions* et les fonctions booléennes des *fonctions d'ordre supérieur*. On définit les termes suivants qui représentent « vrai », « faux » et la fonction « et » :

— $\mathbf{T} = \lambda x.\lambda y.x$
— $\mathbf{F} = \lambda x.\lambda y.y$
— $\mathbf{AND} = \lambda x.\lambda y.((x\ y)\ \mathbf{F})$, soit $\lambda x.\lambda y.x\ y\ \mathbf{F}$ avec les parenthèses implicites

1. Vérifier les réductions suivantes

$$\begin{aligned} \mathbf{AND}(\mathbf{T})(\mathbf{T}) &\rightsquigarrow \mathbf{T} \\ \mathbf{AND}(\mathbf{T})(\mathbf{F}) &\rightsquigarrow \mathbf{F} \\ \mathbf{AND}(\mathbf{F})(\mathbf{T}) &\rightsquigarrow \mathbf{F} \\ \mathbf{AND}(\mathbf{F})(\mathbf{F}) &\rightsquigarrow \mathbf{F} \end{aligned}$$

2. Quelle est la fonction booléenne associée à $\mathbf{XXX} = \lambda x.\lambda y.x\ y\ x$?
3. Qu'est ce qui se passe quand on évalue $\mathbf{AND}\ x\ y$ où x ou y n'est *pas* la représentation d'un booléen ?
4. Donner une représentation de la fonction booléenne « non ».
5. Donner une représentation de la fonction booléenne « ou ».
6. Montrer que le terme $\mathbf{IF} = \lambda x.x$ permet de représenter l'instruction conditionnelle `IF cond THEN M ELSE N`.
7. Coder la représentation des booléens en JavaScript et écrire une fonction qui va afficher les tables de vérité des opérations binaires définies précédemment
8. Écrire une fonction JavaScript qui va comparer si deux opérations codées en λ-calcul ont la même table de vérité.

Corrections

Solution de l'exercice 1

1. 1. Typage :

$$\underbrace{\lambda x.\ \underbrace{\lambda y.\underbrace{(x+y)}_{\texttt{number}}}_{\texttt{number}\to\texttt{number}}\ \underbrace{3}_{\texttt{number}}\ \underbrace{4}_{\texttt{number}}}_{\texttt{number}}$$

   avec x : `number` et y : `number`.

   Évaluation :

$$\lambda x.\lambda y.(x+y)\ 3\ 4 \overset{x}{\rightsquigarrow} \lambda y.(3+y)\ 4 \overset{y}{\rightsquigarrow} 3+4 \rightsquigarrow 7$$

2. Typage : l'expression $(\lambda x.\lambda y.(x\ y))\ 3\ 4$ n'est pas typable car la variable x doit à la fois être un entier (car on passe 3 en paramètre à $(\lambda x.\lambda y.(x+y))$) et une fonction car x est appliquées à y. Par ailleurs, si on essaie d'effectuer les β-reductions, on obtient $(3\ 4)$ qui n'est pas une valeur : seuls les entiers et les fonctions (*i.e.* les expressions commençant par λ) sont des valeurs.
3. Typage : l'expression $\lambda x.(x\ x)$ n'est pas typable. Si on dit que le type de x est un certain α, alors x doit prendre quelque chose de type α en argument (vu qu'on applique x à lui-même). Le type de x doit ainsi être solution de l'équation $\alpha \to \alpha' = \alpha$, *i.e.* en replaçant α à gauche : $(\alpha \to \alpha') \to \alpha' = \alpha$, dans laquelle on peut encore remplacer α à gauche autant de fois qu'on veut.
4. Typage :

$$\underbrace{(\lambda x.\ \lambda y.(\ \underbrace{\underbrace{y}_{\texttt{number}\to\texttt{number}}\ \underbrace{(x+2)}_{\texttt{number}}}_{\texttt{number}}))\ \underbrace{5}_{\texttt{number}}\ (\ \underbrace{\lambda z.(\underbrace{z*3}_{\texttt{number}})}_{\texttt{number}\to\texttt{number}}\ )}_{\texttt{number}}$$

   avec x : `number`, y : `number` $\to$ `number` et z : `number`.

   Évaluation :

$$(\lambda x.\lambda y.(y\ (x+2)))\ 5\ (\lambda z.(z*3)) \overset{x}{\rightsquigarrow} (\lambda y.(y\ (5+2)))(\lambda z.(z*3))$$

$$\overset{y}{\rightsquigarrow} (\lambda z.(z*3))(5+2) \overset{z}{\rightsquigarrow} (5+2)*3 \rightsquigarrow 21$$

   En JavaScript `(x => y => y(x + 2))(5)(z => z *3)`

5. Typage :

$$\underbrace{(\lambda x.((\ \underbrace{\lambda y.\underbrace{(x+y)}_{\texttt{number}}}_{\texttt{number}\to\texttt{number}}\ )\ x))}_{\texttt{number}\to\texttt{number}}\ \underbrace{5}_{\texttt{number}}$$

Évaluation, 2 stratégies possibles :

$$(\lambda x.((\lambda y.(x+y))\ x))\ 5 \ \overset{x}{\rightsquigarrow}\ ((\lambda y.(5+y))\ 5)\ \overset{y}{\rightsquigarrow}\ 5+5\ \rightsquigarrow\ 10$$

et

$$(\lambda x.((\lambda y.(x+y))\ x))\ 5 \ \overset{y}{\rightsquigarrow}\ (\lambda x.(x+x))\ 5\ \overset{x}{\rightsquigarrow}\ 5+5\ \rightsquigarrow\ 10$$

C'est la première qui correspond le plus à JavaScript où on évalue d'abord les arguments.

6. Typage :

$$\underbrace{\underbrace{(\lambda x.(\underbrace{\underbrace{(x\ (\underbrace{\lambda y.(\underbrace{y+2}_{\texttt{number}})}_{\texttt{number}\to\texttt{number}}))}_{\texttt{number}} + \underbrace{(x\ (\underbrace{\lambda z.(\underbrace{z*3}_{\texttt{number}})}_{\texttt{number}\to\texttt{number}})))}_{\texttt{number}}}_{\texttt{number}}))}_{((\texttt{number}\to\texttt{number})\to\texttt{number})\to\texttt{number}}\quad \underbrace{((\underbrace{\lambda u.\ \underbrace{\lambda w.(\underbrace{w\ u}_{\texttt{number}})}_{(\texttt{number}\to\texttt{number})\to\texttt{number}}}_{\texttt{number}\to(\texttt{number}\to\texttt{number})\to\texttt{number}})\ \underbrace{5}_{\texttt{number}})}_{(\texttt{number}\to\texttt{number})\to\texttt{number}}}_{\texttt{number}}$$

avec x : $(\texttt{number} \to \texttt{number}) \to \texttt{number}$, y :, z : $\texttt{number}$, u : $\texttt{number}$, w : $\texttt{number} \to \texttt{number}$

Évaluation :

$$(\lambda x.((x\ (\lambda y.(y+2))) + (x\ (\lambda z.(z*3)))))\quad ((\lambda u.\lambda w.(w\ u))\ 5)$$

$$\overset{u}{\rightsquigarrow}\ (\lambda x.((x\ (\lambda y.(y+2))) + (x\ (\lambda z.(z*3)))))\quad (\lambda w.(w\ 5))$$

$$\overset{x}{\rightsquigarrow}\ (((\lambda w_1.(w_1\ 5))\ (\lambda y.(y+2))) + ((\lambda w_2.(w_2\ 5))\ (\lambda z.(z*3))))$$

$$\overset{w_1}{\rightsquigarrow}\ (((\lambda y.(y+2))\ 5) + ((\lambda w_2.(w_2\ 5))\ (\lambda z.(z*3))))$$

$$\overset{y}{\rightsquigarrow}\ ((5+2) + ((\lambda w_2.(w_2\ 5))\ (\lambda z.(z*3))))$$

$$\overset{w_2}{\rightsquigarrow}\ ((5+2) + ((\lambda z.(z*3))\ 5))$$

$$\overset{z}{\rightsquigarrow}\ ((5+2) + (5*3))\ \rightsquigarrow\ 22$$

Remarque : ici, on a distingué les 2 occurrences de w en renommant w en w_1 et w_2 pour montrer leur indépendance. Si on ne fait que des β-reduction, il ne faudrait pas mettre d'indice, mais on perd en clarté.

Pour le vérifier le résultat en JavaScript :

```
(x => (x(y => y+2) + x(z => z*3))) ((u => w => w(u))(5))
```

7. $(\lambda x.\lambda y.(x\ y))(\lambda z.z) :: \texttt{number} \to \texttt{number}$
   $(\lambda x.\lambda y.(x\ y))(\lambda z.z)\ \overset{x}{\rightsquigarrow}\ \lambda y.((\lambda z.z)\ y)\ \overset{y}{\rightsquigarrow}\ \lambda y.y$

8. $\lambda x.(x+3)(\lambda y.y)$ n'est pas typable
   $\lambda x.(x+3)(\lambda y.y)\ \overset{x}{\rightsquigarrow}\ (\lambda y.y)+3$

9. $(\lambda x.\lambda y.x) :: \texttt{number} \to \texttt{number} \to \texttt{number}$
   l'expression est déjà β-réduite

10. $(\lambda x.\lambda y.x)\ 3 :: \texttt{number} \to \texttt{number}$
    $(\lambda x.\lambda y.x)\ 3\ \overset{x}{\rightsquigarrow}\ \lambda y.3$

11. $(\lambda x.\lambda y.(y\ (3+x)))\ 4\ (\lambda z.(z*2)) :: \texttt{number}$
    $(\lambda x.\lambda y.(y\ (3+x)))\ 4\ (\lambda z.(z*2))\ \overset{x}{\rightsquigarrow}\ \lambda y.(y\ (3+4))\ (\lambda z.(z*2))\ \overset{y}{\rightsquigarrow}\ (\lambda z.(z*2))\ (3+4)\ \rightsquigarrow\ (\lambda z.(z*2))\ (7)\ \overset{z}{\rightsquigarrow}\ (7*2)\ \rightsquigarrow\ 14$

12. $(\lambda x.((\lambda y.(y*x))\ x))\ 5 :: \texttt{number}$
    $(\lambda x.((\lambda y.(y*x))\ x))\ 5\ \overset{x}{\rightsquigarrow}\ (\lambda y.(y*5))\ 5\ \overset{y}{\rightsquigarrow}\ 5*5\ \rightsquigarrow\ 25$

2. — $(\lambda x.x+5)((\lambda z.z)\ 2) \overset{z}{\rightsquigarrow} (\lambda x.x+5)(2) \overset{x}{\rightsquigarrow} 2+5 \rightsquigarrow 7$
   — $(\lambda x.x+5)((\lambda z.z)\ 2) \overset{x}{\rightsquigarrow} ((\lambda z.z)\ 2)+5 \overset{z}{\rightsquigarrow} 2+5 \rightsquigarrow 7$

Solution de l'exercice 2

1. — `let ex1 = f => x => f(f(x));`
   — `let ex2 = x => y => (x(y))(y);` ou `x => y => x(y)(y)`, attention aux parenthèses de l'application qui diffèrent entre JavaScript et le λ-calcul.
   — `let I = x => x;`
   — `let K = x => y => x;`
   — `let S = x => y => z => (x(z))(y(z));`
   On peut montrer les différentes écritures des fonctions en JavaScript

```
1  function twice1(f){
2    function r(x){
3      return f(f(x));
4    }
5    return r;
6  }
7
8  const twice2 =
9  function (f){
10   return (function (x)
11     {
12       return f(f(x));
13     }
14   )
15 }
```

   On peut inférer la stratégie d'évaluation de JavaScript en évaluant `K(3)(bot)`. Ceci va produire une erreur de type `ReferenceError: bot is not defined` alors que quelque soit la valeur de `bot` l'expression devrait pouvoir se réduire en la valeur 3. Cet exemple montre que l'interpréteur a bien essayé d'évaluer le deuxième argument y de **K** (stratégie *eager*).
2. On peut définir $\mathbf{I} = \mathbf{SKK}$ et vérifier que $\mathbf{I}\,x$ et $\mathbf{SKK}\,x$ se réduisent dans le même terme x. On va rajouter une étape d'α-renommage pour que ce soit plus lisible.

$$\begin{aligned}
\mathbf{SKK}x &= (\lambda x.\lambda y.\lambda z.(x\,z)(y\,z))(\lambda x.\lambda y.x)(\lambda x.\lambda y.x)x \\
&\overset{\alpha}{\rightsquigarrow} (\lambda x_0.\lambda y_0.\lambda z_0.(x_0\,z_0)(y_0\,z_0))(\lambda x_1.\lambda y_1.x_1)(\lambda x_2.\lambda y_2.x_2)x \\
&\overset{x_0}{\rightsquigarrow} (\lambda y_0.\lambda z_0.((\lambda x_1.\lambda y_1.x_1)z)(y_0\,z_0))(\lambda x_2.\lambda y_2.x_2)x \\
&\overset{y_0}{\rightsquigarrow} (\lambda z_0.((\lambda x_1.\lambda y_1.x_1)z_0)((\lambda x_2.\lambda y_2.x_2)z_0))x \\
&\overset{x_1}{\rightsquigarrow} (\lambda z_0.(\lambda y_1.z_0)((\lambda x_2.\lambda y_2.x_2)z_0))x \\
&\overset{x_2}{\rightsquigarrow} (\lambda z_0.(\lambda y_1.z_0)(\lambda y_2.z_0))x \\
&\overset{y_1}{\rightsquigarrow} (\lambda z_0.z_0)x \\
&\overset{z_0}{\rightsquigarrow} x
\end{aligned}$$

3. On remarquera au passage que `fn1` est la composition fonctionnelle notée usuellement $(f \circ g)$
   — $\lambda f.\lambda g.\lambda x.f\ (g\ x)$
   — $\lambda f.\lambda g.\lambda x.(g\ x)+(f\ x)$
4. Le résultat après réduction est $2+(2*3)$ soit 8, en JavaScript `(f => g => x => f(g(x)))(x => 2 + x)(x => 2 *x)(3)`
5. On obtient `x => 2 + (2 * x)` d'après les réductions suivantes :

$$
\begin{aligned}
& (\lambda f.\lambda g.\lambda x.f(gx))(\lambda y.2+y)(\lambda z.2*z) \\
\rightsquigarrow\ & (\lambda g.\lambda x.(\lambda y.2+y)(gx))(\lambda z.2*z) \\
\rightsquigarrow\ & (\lambda x.(\lambda y.2+y)((\lambda z.2*z)x)) \\
\rightsquigarrow\ & (\lambda x.(\lambda y.2+y)(2*x)) \\
\rightsquigarrow\ & (\lambda x.2+(2*x))
\end{aligned}
$$

6. On peut faire la démonstration dans la console JavaScript
   — `fn3(3)` `TypeError: x is not a function`
   — `fn3(x => x)` `function ()`
   — `fn3(x => x)(3)= 3`
   — `fn3(fn3(x => x))(3)= 3`
7. C'est une question pour rigoler, on peut la passer. `w(x => x +1)` se réduit en `(x => x +1) (x => x +1)` puis en `(x => x +1)+1`. Le `+` à l'extérieur ne peut pas être une addition car le membre gauche est la fonction `x => x +1`. Le terme `x => x +1` est alors *implicitement converti en chaîne de caractère* `"x => x +1"` tout comme le membre droit qui devient `"1"` et l'opération + devient la concaténation de `"x => x +1"` et `"1"` pour le résultat obtenu.

Solution de l'exercice 3

On peut expliquer, en guise d'ouverture, qu'on peut faire un codage similaire avec les entiers.

1. En remarquant que $\mathbf{T}(Z) \rightsquigarrow \lambda y.Z$ et que $\mathbf{F}(Z) \rightsquigarrow \lambda y.y$ on peut obtenir les dérivations suivantes assez rapidement.

$$
\begin{aligned}
\mathbf{AND}(\mathbf{T})(\mathbf{T}) &\rightsquigarrow (\lambda x.\lambda y.x\ y\ \mathbf{F})(\mathbf{T})(\mathbf{T}) \\
&\overset{x}{\rightsquigarrow} (\lambda y.\mathbf{T}\ y\ \mathbf{F})(\mathbf{T}) \\
&\overset{y}{\rightsquigarrow} (\mathbf{T}\ \mathbf{T})\ \mathbf{F} \\
&\rightsquigarrow ((\lambda x.\lambda y.x)(\lambda x.\lambda y.x))\ \mathbf{F} \\
&\overset{x}{\rightsquigarrow} (\lambda y.(\lambda x.\lambda y.x))\ \mathbf{F} \\
&\overset{y}{\rightsquigarrow} (\lambda x.\lambda y.x) \\
&\rightsquigarrow \mathbf{T} \\
\mathbf{AND}(\mathbf{T})(\mathbf{F}) &\rightsquigarrow ((\lambda x.\lambda y.x)(\lambda x.\lambda y.y))\ \mathbf{F} \\
&\overset{x}{\rightsquigarrow} (\lambda y.(\lambda x.\lambda y.y))\ \mathbf{F} \\
&\overset{y}{\rightsquigarrow} (\lambda x.\lambda y.y) \\
&\rightsquigarrow \mathbf{F} \\
\mathbf{AND}(\mathbf{F})(\mathbf{T}) &\rightsquigarrow ((\lambda x.\lambda y.y)(\lambda x.\lambda y.x))\ \mathbf{F} \\
&\overset{x}{\rightsquigarrow} (\lambda y.y)\ \mathbf{F} \\
&\overset{y}{\rightsquigarrow} \mathbf{F} \\
\mathbf{AND}(\mathbf{F})(\mathbf{F}) &\rightsquigarrow ((\lambda x.\lambda y.y)(\lambda x.\lambda y.y))\ \mathbf{F} \\
&\overset{x}{\rightsquigarrow} (\lambda y.y)\ \mathbf{F} \\
&\overset{y}{\rightsquigarrow} \mathbf{F}
\end{aligned}
$$

2. On va évaluer les quatre cas $\mathbf{XXX}(\mathbf{F})(\mathbf{F})$, $\mathbf{XXX}(\mathbf{T})(\mathbf{F})$, $\mathbf{XXX}(\mathbf{F})(\mathbf{T})$ et $\mathbf{XXX}(\mathbf{T})(\mathbf{T})$ pour remarquer qu'il s'agit d'une autre expression du « et ».

3. Il y aura bien réduction mais le résultat ne sera pas nécessairement la représentation d'un booléen, par exemple $\mathbf{AND}(\lambda x.x)(\lambda x.x\ x)$ se réduit en **FF** puis en $\lambda y.y$ qui ne représente pas un booléen : *garbage in, garbage out* comme on dit.
4. On peut proposer $\mathbf{NOT} = \lambda x.x\ \mathbf{F}\ \mathbf{T}$, l'intuition étant que pour inverser les valeurs booléennes, on échange les deux derniers arguments passés à **NOT**.
5. On peut proposer, parmi plusieurs solutions possibles, $\mathbf{OR} = \lambda x.\lambda y.y\ y\ x$ ou $\mathbf{OR} = \lambda x.\lambda y.x\ \mathbf{T}\ y$. L'expression $\lambda x.\lambda y.x\ y\ \mathbf{T}$ représente quant à elle l'implication.
6. On va montrer que **IF T** M N se réduit en M et que **IF F** M N se réduit en N ce qui capture l'idée de l'instruction conditionnelle. On donne le premier cas, le second étant similaire.

$$\begin{aligned}\mathbf{IF}\ \mathbf{T}\ M\ N &\rightsquigarrow ((((\lambda x.x)(\mathbf{T}))M)N)\\ &\rightsquigarrow (((\mathbf{T})M)N)\\ &\rightsquigarrow M\end{aligned}$$

7. Pour rendre les booléens de Church affichables, on doit passer des valeurs aux paramètres x et y. Par exemple, on définira `let T = x => y => x` et on appellera `T(true)(false)` pour avoir une valeur affichable, `T` affichant simplement `function T()`. L'exemple complet est donné ci-après.

```
1
2
3  let T = x => y => x;           //Church encoding of TRUE
4  let F = x => y => y;           //Church encoding of FALSE
5
6  let AND1 = x => y => x(y)(F); //Church encoding of AND, 1st ver
7  let AND2 = x => y => x(y)(x); //Church encoding of AND, 2nd ver
8
9  let NOT = p => p(F)(T);        //Church encoding of NOT
10
11 let OR1 = x => y => y(T)(x);  //Church encoding of OR, 1st ver
12 let OR2 = x => y => y(y)(x);  //Church encoding of OR, 2nd ver
13 let OR3 = x => y => NOT(AND1(NOT(x))(NOT(y)));    //De Morgan
14 let OR4 = x => y => (x(F)(T))(y(F)(T))(F)(F)(T);  //OR3 beta
15
16 let IMP = x => y => x(y)(T);  //Church encoding of IMPLIES
17
18 let bools = [T, F];
19 let ev = f => f("true")("false"); //gives concrete arguments
20
21 let show_truth_table =  op =>
22   bools.map(v1 =>
23     bools.map(v2 =>
24       console.log(ev(v1) + '\t' + ev(v2)  + '\t' + ev(op(v1)(v2)))
25     )
26   );
27
28 //with map and reduce only, for fun
29 let compare_truth_table =  op1 => op2 =>
30   values.map(v1 =>
31     values.map(v2 => (ev(op1(v1)(v2)) == ev(op2(v1)(v2))))
32     ).map(t => t.reduce((a,c) => a && c , true))
33      .reduce((a,c) => a && c , true) ;
34
35 function main(){
36   console.log("***Truth table unary NOT***");
37   values.map(v => console.log(ev(v) + '\t' + ev(NOT(v))));
38
39   for(op of [AND1, OR1, OR4, IMP]){
40     console.log("***Truth table binary " + op.name + " ***" );
41     show_truth_table(op);
42   }
43
44   console.log("***Check AND1 == AND2***");
45   console.log(compare_truth_table(AND1)(AND2));
46   console.log("***Check AND1 == IMP***");
47   console.log(compare_truth_table(AND1)(IMP));
48   console.log("***Check OR1 == OR3***");
49   console.log(compare_truth_table(OR1)(OR3));
50   console.log("***Check OR1 == OR4***");
51   console.log(compare_truth_table(OR1)(OR4));
52
53 }
54
55 main();
```