TD5 de compilation
Clôtures

Exercice 1 (Classe inversée : typage dynamique).

1. Quel est le résultat de la ligne suivante en JS ?

   ```
   2 + True >= (2 && False) ;
   ```

2. Écrivez un code assembleur mini-JS-machine correcte.
3. Écrivez un code pour la ligne suivante qui soit correcte que x soit un entier ou un Booléen :

   ```
   If (x) y = False; Else y = x+1;
   ```

   On a le droit aux instructions `TypeOf` et `Case`.
4. Quel est le résultat de la ligne suivante en JS ? (Attention : les strings ne sont pas dans le projet cette année)

   ```
   2 + " pièges" >= 1e3 + " test" ;
   ```

5. et en remplaçant `1e3` par `1e1000` ?
6. Écrivez un code pour la ligne suivante qui soit correcte que y et z soient des entiers, Booléen ou String :

   ```
   x = y + z;
   ```

   On a le droit à l'instruction `Swap`.

Dans les différents codes vus ci-dessous, on suppose toujours que l'on connait statiquement le type des variables, affin de ne pas avoir à écrire tous ces tests dynamiques.

Exercice 2 (Clôtures simples).

1. Écrivez un code assembleur mini-JS-machine correcte pour le code suivant (où `x` et `y` sont des nombres) :

   ```
   z = f(x+y) == 3;
   function f (x) {
     return x - y;
   }
   ```

2. Écrivez un autre code correcte sur la même expression.
3. Simulez un code en supposant que x et y valent 2 au départ.

Exercice 3 (Décompilation).
Voici un code assembleur :

```
DecVar f
NewClo 18
DecArg x
DecArg y
SetVar f
GetVar f
StCall
CsteNb 10
Log
SetArg
GetVar f
StCall
CsteNb 30
SetArg
CsteNb 8
SetArg
Call
SetArg
Call
Halt
#commentaire ?
DecVar g
NewClot 25
DecArg x
SetVar g
GetVar x
CsteNb 0
GrStNb
Log
ConJmp 14
GetVar f
StCall
GetVar x
GetVar y
SubiNb
SetArg
GetVar g
StCall
GetVar y
SetArg
Call
SetArg
Call
Return
GetVar y
GetVar x
AddiNb
Return
GetVar x
CstNb 1
AddiNb
Log
Return
```

1. Exécutez le (pas besoin d'aller jusqu'au bout) en écrivant les logs.
2. Décompilez-le en ignorant les instructions `Log`.

Exercice 4 (Ordre supérieur).
Écrivez un code assembleur mini-JS-machine correcte pour le code suivant (où `x` et `y` sont des nombres) :

```
f (g) (4)
function g (x) { return x+x;}
function f (g) {
  if (g(5)>30) return compose(g,g)
  return compose(f,f)(compose(g,g))
}
function compose (f,g) {
  return h;
  function h (x) { return f(g(x))}
}
```

Instruction	sémantique	pile avant	pile après
Log	`Print(Pull);`	`X:pile`	`X:pile`
AddiNb, SubsNb, MultNb, DiviNb	`Push(Pop` $\odot_f$ `Pop);`	`#n:#n:pile`	`#n:pile`
NegaNb	`Push(Pop` $*_f$ (-1)`);`	`#n:pile`	`#n:pile`
BoToNb, NbToBo NbToSt, StToNb	cast du sommet	`X:pile`	`Y:pile`
NbToBo	`DoubleToBool(Pop);`	`#n:pile`	`#b:pile`
NbToSt	`DoubleToString(Pop);`	`#n:pile`	`#s:pile`
Equal	`Push(Pop == Pop);`	`#v:#v:pile`	`#b:pile`
LoEqNb	`Push(Pop` $\leq_f$ `Pop);`	`#n:#n:pile`	`#b:pile`
GrStNb	`Push(Pop > Pop);`	`#n:#n:pile`	`#b:pile`
Jump offset	`PC := PC + off + 1;`	`pile`	`pile`
ConJmp offset	if `Pop` then `PC := PC + 1;` else `PC := PC + off + 1;`	`#b:pile`	`pile`
Case p	`PC := PC + p * Floor(Pop) + 1;`	`#n:pile`	`pile`
CstNb x	`Push(x);`	`pile`	`#f:pile`
CstBo x	`Push(x);`	`pile`	`#b:pile`
Copy	`Push(Pull);`	`X:pile`	`X:X:pile`

Swap	`#vx := Pop;`
	`#vy := Pop;`
	`Push(x);`
	`Push(y);`
		`X:Y:pile`	`Y:X:pile`

Drop	`Pop;`	`X:pile`	`pile`
TypeOf	Pull une valeur, rend 0 sur un booleen, 1 sur un flottant, 2 sur une string, 3 sur undefined, 4 sur une cloture, 5 sur un objet,	`X:pile`	`#n:X:pile`
SetVar n	`Set(n, Pull);`	`#v:pile`	`pile`
GetVar n	`Push(Get(n))`	`pile`	`#v:pile`
Concat	`Push(Pop` $+_s$ `Pop);`	`#s:#s:pile`	`#s:pile`
CstStr x	`Push(x);`	`pile`	`#s:pile`
StToNb	`StringToDouble(Pop);`	`#s:pile`	`#n:pile`
NbToSt	`DoubleToString(Pop);`	`#n:pile`	`#s:pile`
DclVar n	`Insert(n, undefind)`	`pile`	`pile`
NewClo off	`Push(NewCloture{cont = CopyCont,` `code = PC + off + 1})`	`pile`	`#l:pile`
DclArg n	`Pull.args.Push(n)`	`#l:pile`	`#l:pile`
StartCall	`Pull.setContext(NewContext(CC))`	`#l:pile`	`#l:pile`

SetArg	`#v v = Pop`
	`#l clot = Pull`
	`#n n = clot.args.Pop`
	`clot.cont.Insert(n, v)`
	`PC := PC + 1`
		`#v:#c:pile`	`#c:pile`

Call	`#l clot = Pop`
	`Push(NewContinuation{cont = CC,`
	`code = PC,`
	`err = 0})`
	`CC := clot.cont`
	`PC := clot.code`
		`#l:pile`	`#t:pile`

Return	`#v res = Pop;`
	`do {#t continue = Pop;}`
	`while (continue.err)`
	`CC := continue.cont`
	`PC := continue.code`
	`Push(res)`
		`X:autre@#t:pile`	`X:pile`