TD5 de compilation
Typage dynamique

Exercice 1 : 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 en TD par la suite, 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.

--- * ---

Instruction	sémantique	pile avant	pile après
Log	`Print(Pull);`	v::stk	v::stk
AddiNb, SubsNb, MultNb, DiviNb	`Push(Pop` $\odot_f$ `Pop);`	n_1::n_2::stk	n_3::stk
NegaNb	`Push(Pop` $*_f$ (-1)`);`	n_1::stk	n_2::stk
NbToBo	`DoubleToBool(Pop);`	n::stk	b::stk
NbToSt	`DoubleToString(Pop);`	n::stk	`#s:pile`
BoToNb, StToBo BoToSt, StToNb	cast du sommet	v_1::stk	v_2::stk
Equal	`Push(Pop == Pop);`	v_1::v_2::stk	b::stk
LoEqNb	`Push(Pop` $\leq_f$ `Pop);`	n_1::n_2::stk	b::stk
GrStNb	`Push(Pop > Pop);`	n_1::n_2::stk	b::stk
Jump offset	`PC := PC + off + 1;`	stk	stk
ConJmp offset	if Pop then `PC := PC + 1;` else `PC := PC + off + 1;`	b::stk	stk
Case p	`PC := PC + p * Floor(Pop) + 1;`	n::stk	stk
CstNb x	`Push(x);`	stk	n::stk
CstBo x	`Push(x);`	stk	b::stk
Copy	`Push(Pull);`	v::stk	v::v::stk

Swap	`#vx := Pop;`
	`#vy := Pop;`
	`Push(x);`
	`Push(y);`
		v_1::v_2::stk	v_2::v_1::stk

Drop	`Pop;`	v::stk	stk
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,	v::stk	n::v::stk
SetVar n	`Set(n, Pull);`	v::stk	stk
GetVar n	`Push(Get(n))`	stk	v::stk
Concat	`Push(Pop` $+_s$ `Pop);`	s_1::s_2::stk	s_3::stk
CstStr x	`Push(x);`	stk	s::stk
StToNb	`StringToDouble(Pop);`	s::stk	n::stk
NbToSt	`DoubleToString(Pop);`	n::stk	s::stk
DclVar n	`Insert(n, undefind)`	stk	stk
NewClo off	`Push(NewCloture{cont = CopyCont,` `code = PC + off + 1})`	stk	l::stk
DclArg n	`Pull.args.Push(n)`	l::stk	l::stk
StartCall	`Pull.setContext(NewContext(CC))`	l::stk	l::stk

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

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

Return	`#v res = Pop;`
	`do {#t continue = Pop;}`
	`while (continue.err)`
	`CC := continue.cont`
	`PC := continue.code`
	`Push(res)`
		v::$\cdots$::t::stk	v::stk