Examen de Compilation
première session 2022

Exercice 1 (cours)**.** Répondez à la question suivante (moins d'une ligne) :

1. Sur votre langage compilé préféré, quelle commande/programme utilisez vous comme compilateur ?
2. Qu'est-ce qu'une expression régulière ?
3. Qu'attend-on en sortie du lexeur ?
4. Qu'attend-on en sortie du générateur de parseur ?
5. Qu'est-ce que l'erreur *"stack overflow"* ?

Répondez aux la questions suivantes (2-3 lignes par réponses, pas plus) :

6. Que sont les prototypes ?
7. Comment propose-t-on de les implémenter dans le cours ?

Exercice 2 (cours+prog)**.** Dans le langage de votre choix, écrivez :

1. une structure pour les automates shift-actions,
2. une fonction d'évaluation d'un mot sur un tel automate,
3. ainsi qu'une fonction de psedo-déterminisation.

Essayez d'être aussi général que possible, si vous n'y arrivez pas, choisissez une restriction "raisonnable" que vous préciserez en français.

Exercice 3 (typage dynamique)**.** Écrivez un code assembleur pour le sous-programme:

```
if (x) then x=y+2;
```

qui soit correcte que `x` et `y` soient un nombre, un Booléen ou un undefined.

Exercice 4 (parseur)**.** Voici une grammaire dont les terminaux sont a, b et c

$S := RaS \mid \epsilon$
$R := Rb \mid Tc \mid \epsilon$
$T := Tc \mid \epsilon$

1. Écrire un mot de plus de 3 lettres reconnu par cette grammaire.
2. Construire le parseur SLR.

Exercice 5 (éxécution)**.** Voici un programme dans l'assembleur vu en cours; on suppose que seule la variable `ine` est initialisée avec votre numéro étudiant lors de l'entrée dans le programme, les autres ne sont pas initialisées. L' instruction `Log` copie la tête de pile (sans la retirer) dans une trace. Donnez la trace écrite par les `Log` pendant l'exécution du programme (et uniquement cette trace).

```
# ine : num etu
GetVar ine
GetVar x
Log
Drop
CsteNb 100
CsteNb 10
MultNb
Log
SetVar m
Log
GetVar m
GrStNb
ConJmp 2
CsteNb 42
Log
NewClot 19
StCall
Call
SetVar f
GetVar f
StCall
NewClo 13
StCall
Call
StCall
GetVar f
StCall
GetVar ine
SetArg
Call
SetArg
Call
SetArg
Call
Halt
DecVar u
GetVar m
SetVar u
NewClot 2
DecArg x
Return
GetVar x
GetVar u
Copy
CsteNb 10
MultNb
SetVar u
SubiNb
Log
Return
```

Instruction	sémantique	pile avant	pile après
Log	`Print(Pull)`;	`X:pile`	`X:pile`
AddiNb, SubsNb, MultNb, DiviNb	`Push(Pop` $\odot_f$ `Pop)`; avec $\odot$ représentant, respectivement, $+$, $-$, $*$ et $/$	`#n:#n:pile`	`#n:pile`
GeStNb	`Push(Pop` $>_f$ `Pop)`;	`#n:#n:pile`	`#b:pile`
CsteNb x	`Push(x)`;	`pile`	`#n:pile`
Copy	`Push(Pull)`;	`X:pile`	`X:X:pile`
Drop	`Pop()`;	`X:pile`	`pile`
GetVar n	`Push(Get(n))`	`pile`	`#v:pile`
SetVar n	`Set(n, Pop())`	`pile`	`#v:pile`
DclVar n	`Insert(n, undefind)`	`pile`	`pile`
NewClo off	`Push(NewCloture{cont := CopyCont, code := PC+off+1})`	`pile`	`#l: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`
DclArg n	`Pull.args.Push(n)`	`#l:pile`	`#l:pile`
StCall	`Pull.setContext(NewContext(CC))`	`#l:pile`	`#l:pile`
SetArg	`v := Pop; clot := Pull; n := clot.args.Pop;` `clot.cont.Insert(n, v);`	`#v:#c:pile`	`#c:pile`
Call	`clot := Pop; Push(NewContinuation{cont := CC, code := PC})` `CC := clot.cont; PC := clot.code`	`#l:pile`	`#t:pile`
Return	`res := Pop; continue := Pop;` `CC := continue.cont; PC := continue.code; Push(res)`	`X:#t:pile`	`X:pile`
BoToNb	`if Pop then Push(1) else Push(0)`	`#b:pile`	`#n:pile`
NbToBo	`Push(Pop` $\neq_f$ `0)`	`#n:pile`	`#b:pile`
Case p	`PC := PC + p * Floor(Pop) + 1`;	`#n:pile`	`pile`
TypeOf	Peak une valeur, rend 0 sur un Boolean, 1 sur un Number, 2 sur un Undefined, ...	`X:pile`	`#n:X:pile`
Halt	Fin du programme (arrête la machine)		