.

Compilation
Examen 2019

Exercice 1. Donnez les différentes étapes de la compilation, en précisant les entrées/sorties de chaque étape.

Donnez la différence entre compilation et interprétation.

Exercice 2. Générez un lexeur pour les tokens suivants :

- commentaire : toute chaîne commençant par `{*` et finissant par `*}`, et qui ne contient pas `*}`,
- les flottants de la forme `12.54`, `.54`, ou `12.`,
- les entiers de caml, avec des underscore dedans (ex: 2_000 ou 06_23_21), mais pas en fin ni en début de nombre, ni deux à la suite,
- avec des espaces et des tab comme séparateurs de token.

Vous pouvez le donner graphiquement sous la forme d'un automate shift-action sans conflit.

Exercice 3. Donnez un encodage correct du programme suivant dans la mini-Py-machine. Vous avez le droit aux optimisations que vous voulez, en particulier, vous n'avez pas a vérifier le type si vous le connaissez. Si vous ne savez pas faire certaines parties du code, vous pouvez les mettre entre crochets et faire le reste. Si vous mettez presque tout entre crochets, vous n'aurez pas beaucoup de points.
Ce code est le corps d'une fonction, on y dispose d'une variable `x` entière présente dans le contexte (l'argument de la fonction).

```
u = f(x+3)
y = 42
Tantque x > 2 and x <= 20 :
  u.ajoute(x)
  y = (x,y)
  x = x-1

Def f (z) :
  x = Liste()
  x.ajoute(z)
  retourne x

Classe Liste :
  x = Null
  ajoute(z) :
    x = (z,x)
```

Exercice 4. La grammaire suivante est-elle LR_0 ? SLR ? LALR ? LR1 ?
$S := XaX \mid Ya$
$X := b \mid aX$
$Y := Xab$
Montrez-le.

Exercice 5. La grammaire suivante est-elle LR_0 ? SLR ? LALR ? LR1 ?
$\langle \texttt{seq} \rangle := \langle \texttt{seq} \rangle @ \langle \texttt{INT} \rangle \mid (\langle \texttt{seq} \rangle) \mid \langle \texttt{INT} \rangle \mid \langle \texttt{INT} \rangle @ \langle \texttt{seq} \rangle$
Montrez-le.

Instruction	sémantique	pile avant	pile après
AddE	Push(Pop $+_e$ Pop);	e:e:pile	e:pile
SubE	Push(Pop $-_e$ Pop);	e:e:pile	e:pile
And	Push(Pop && Pop);	e:e:pile	e:pile
Or	Push(Pop || Pop);	e:e:pile	e:pile
LeE	Push(Pop $\leq_e$ Pop);	e:e:pile	e:pile
GeE	Push(Pop $\geq_e$ Pop);	e:e:pile	e:pile
LsE	Push(Pop $<_e$ Pop);	e:e:pile	e:pile
GsE	Push(Pop $>_e$ Pop);	e:e:pile	e:pile
NewTuple	Push(NewTuple{taille = Pop, tab = newTab(taille)	e:pile	u:pile
GetU	Push(Pop.tab[Pop]);	e:u:pile	X:pile
SetU	Pull.tab[Pop] := Pop;	X:e:u:pile	u:pile
Jump offset	PC := PC + off;	pile	pile
Cjmp offset	if Pop then PC := PC; else PC := PC + off;	e:pile	pile
CstE x	Push(x);	pile	e:pile
Copy	x := Pull; Push(x);	X:pile	X:X:pile
Swap	x := Pop; y := Pop; Push(x);Push(y);	X:Y:pile	Y:X:pile
Drop	Pop;	X:pile	pile
Modi n	if Existe(n) then Modif(n, Pop); else Inserer(n, Pop);	X:pile	pile
Get n	Push(Get(n))	pile	X:pile
Halt	arrête la machine	pile	
TypeVerif n	Push(Pop.nom $==_s$ n)	o:pile	e:pile

Instruction	sémantique	pile avant	pile après
AjoutVide n	Insert(n, Null)	pile	pile
Lambda off	Push(NewCloture{cont = CopyLock, methode = PC + off + 1, args = NewPile})	pile	l:pile
DeclA n	Pull.args.Push(n)	l:pile	l:pile
Args	v := Pop clot := Pop n := clot.args.Pop clot.cont := clot.cont.Insert(n, v) Push(clot)	X:c:pile	c:pile
Appel	clot = Pop Assert(IsEmpty(clot.args)) Push(CC) Push(PC) CC := clot.cont PC := clot.methode	l:pile	a:a:pile
Return	res := Pop PC := Pop CC := Pop Push(res)	X:a:a:pile	X:pile
Point n	Push(Pop.cont.Get(n))	o:pile	X:pile
Constr	Push(Pop.cont.Get("__init__"))	t:pile	l:pile
NewObj n	Push(NewObjet{classe = n, cont = Null})	pile	o:pile
NewClass n	Push(NewObjet{classe = "Classe", cont = NewCont{ nom = "__init__", val = Null, cont = Null}}	pile	t:pile
AjouterConstr	Pull.cont.Inserer("__init__", Pop)	l:t:pile	t:pile
ModiObj n	Pop.cont.Modi(n, Pop)	X:t:pile	pile