Examen de Compilation
première session 2025

Exercice 1 (cours, 4pt, 20min)**.** Réponses courtes (1-2 lignes) :

1. Qu'est-ce que le typage dynamique ?
2. Qu'est-ce que PC (program counter) ?
3. Qu'attend-on en sortie du backend ?
4. Qu'attend-on en entrée du générateur de parseur ?

Réponse détailées (4-8 lignes) :

5. Qu'est-ce que l'annalyse lexical ?
6. Quelles sont les 3 manières de compiler les exceptions ?

Exercice 2 (Lexer+prog, 4pt, 50min)**.** 1) Écrivez une grammaire, puis,
2) écrivez un fichier de génération de parseur dans le langage de votre choix (bison, yacc, ocamlyacc, javacc, ect...) qui :

- suppose, depuis le lexeur que l'on n'a pas besoin d'écrire, comme token, les mots clés `SELECT`, `FROM`, `WHERE`, `AND`, `OR` et `IN` ainsi que les opérateurs `,`, `.`, `=` et `>`, les parenthèses, les tokens `Int` et `Id` des identifiants et nombres,
- reconnaisse la sous-grammaire de SQL permettant de faire des requettes en lecture utilisant ces mots-clés et opérations.

On sera très laxiste sur la syntaxe exacte (du lanuage choisi comme de SQL), les bibliothèques et la gestion des erreurs, par contre la structure devra être respectée et les opérateur utilisés doivent exister (avec potentielement une syntaxe légèrement différente).

Voici quatre exemples :

- ```
  SELECT nom, service, enseignant.statut
  FROM   enseignant
  WHERE  enseignant.annee = 2025
  AND (indice > enseignant.categorie  OR categorie = 12)
  ```
- ```
  SELECT nom, service, enseignant.statut
  FROM   enseignant
  ```
- ```
  SELECT nom, service, enseignant.statut
  FROM   enseignant
  WHERE  service.annee IN
  (  SELECT formation.annee
     FROM   formation
     WHERE  formation.niveau = 3)
  ```
- ```
  SELECT nom, service, enseignant.statut
  FROM   enseignant
  WHERE  service.annee IN (2022, 2023, 2026)
  ```

Exercice 3 (VM+prog, 4pt, 40min)**.** Écrivez un programme dans le langage de votre choix (ocaml, java, C, ect...) qui implémente une partie de la mini-JS-machine, avec :

1. deux structure de donnée décrivant les clotures et les continuations,
2. une fonction permetant de simuler le comportement de l'instruction `Call`
3. une fonction permetant de simuler le comportement de l'instruction `Throw`

Les structures de données non demandées peuvent être suposées implémentées, avec les fonctions/méthodes permetant de les manipuler. On sera très laxiste sur la syntaxe exacte.

Exercice 4 (parseur, 4pt, 40min)**.** Voici une grammaire
S := A | B | AB
A := (S) | Aa
B := b | bB

1. Quels sont les non-terminaux ? les terminaux ?
2. Construire le parseur SLR.
3. Indiquer les conflits.

Exercice 5 (Decompilation, 4pt, 30mn)**.** Voici un programme dans l'assembleur vu en cours. Décompilez-le, c'est à dire essayez de retrouver le programme dont il est originel. Vous pouvez ne décompiler que des parties ou laisser des trous, donnez bien les lignes corespondantes aux parties traduites.

```
1  DecVar f
2  DecVar g
3  DecVar y
4  NewClo x
5  SetVar f
6  NewClo g
7  DecArg x
8  SetVar g
9  CsteNb 2
10 SetVar y
11 GetVar f
12 StCall
13 Call
14 Drop
15 GetVar f
16 StCall
17 Call
18 Drop
19 GetVar g
20 StCall
21 GetVar f
22 SetArg
23 Call
24 SetVar x
25 GetVar x
26 StCall
27 Call
28 Drop
29 GetVar x
30 StCall
31 Call
32 Drop
33 Halt
34 x:
35 GetVar y
36 CsteNb 1
37 AddiNb
38 Copy
39 SetVar y
40 Return
41 g:
42 DecVar y
43 CsteNb 0
44 SetVar y
45 NewClo 1
46 Return
47 DecVar i
48 GetVar y
49 SetVar i
50 GetVar i
51 CsteNb 0
52 GrEqNb
53 CndJmp 9
54 GetVar x
55 StCall
56 Call
57 Drop
58 GetVar i
59 CsteNb 1
60 SubsNb
61 SetVar i
62 Jump -13
63 GetVar y
64 CsteNb 1
65 AddiNb
66 SetVar y
67 CsteUn
68 Return
```

Bonus : Devinez la valeur de y à la fin du programme.

Instruction	sémantique	pile avant	pile après
AddiNb	`Push(Pop` $+_{\mathtt{f}}$ `Pop)`;	n_1::n_2::p	n_3::p
SubsNb	`Push(Pop` $-_{\mathtt{f}}$ `Pop)`;	n_1::n_2::p	n_3::p
CstNb x	`Push(x)`;	p	n::p
CstUn	`Push(undefined)`;	p	u::p
GrEqNb	`Push(Pop` $\geq_{\mathtt{f}}$ `Pop)`;	n_1::n_2::p	b::p
Copy	`Push(Pull)`;	v::p	v::v::p
Drop	`Pop()`;	v::p	p
GetVar n	`Push(Get(n))`	p	v::p
SetVar n	`Set(n, Pop())`	v::p	p
DecVar n	`Insert(n, undefind)`	p	p
NewClo lab	nouvelle cloture pointant sur la ligne étiquetée	p	l::p
Jump off	`PC := PC + off + 1;`	p	p
CndJmp off	if `Pop` then `PC := PC + 1;` else `PC := PC + off + 1;`	b::p	p
DclArg n	`Pull.args.Push(n)`	l::p	l::p
StCall	`Pull.setContext(NewContext(CC))`	l::p	l::p
SetArg	`v := Pop; clot := Pull; n := clot.args.Pop;` `clot.cont.Insert(n, v);`	v::l::p	l::p
Call	ouvre la cloture et la remplace par une continuation	l::p	t::p
Return	`res:=Pop; cont:=Pop;` `CC:=cont.cont; PC:=cont.code; Push(res)`	v::t::p	v::p
Throw	va charcher et ouvre la première continuation d'erreur	v::$\cdots$::t::p	v::p
Halt	Fin du programme (arrête la machine)		