Université de Caen **Année 2021/2022**
UFR des Sciences **1ère session**
Lundi 28 mars

L3 – INF6ACT
Théorie des langages et compilation
durée 2h
Documents autorisés : notes personnelles, diapos du cours.

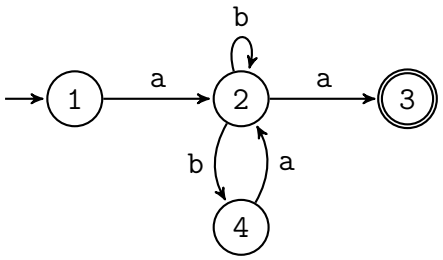
Chaque candidat doit, en début d'épreuve, porter son nom dans le coin de la copie réservé à cet usage; il le cachettera par collage après la signature de la feuille d'émargement. Sur chacune des copies intercalaires, il portera son numéro de place.

Rendre 2 copies séparées en notant bien le numéro de place :
- l'une qui traite l'exercice I (Automate fini) et l'exercice III (analyse LL)
- l'autre qui traite l'exercice II (Compilation)

Exercice I. Automate Fini — À rendre avec l'exercice III

On considère l'automate fini $\mathcal{A}$ décrit par le graphe de transition ci-dessous.

L'automate $\mathcal{A}$



Question 1.

1 . a) Parmi les mots suivants, lesquels sont acceptés par $\mathcal{A}$, et lesquels sont rejetés ?
ε , `aa` , `abba` , `ababa` , `abaaa`

1 . b) Pour chacun des mots acceptés, donner la trace d'un calcul acceptant.

1 . c) Pourquoi l'automate $\mathcal{A}$ est-il non déterministe ?

Question 2. Donner une expression régulière qui décrit le langage reconnu par $\mathcal{A}$.

Question 3. Donner une grammaire qui engendre le langage reconnu par l'automate $\mathcal{A}$.

Question 4. On veut construire un automate $\mathcal{B}$ version déterministe de l'automate $\mathcal{A}$.

4 . a) Construire la table de transition de l'automate $\mathcal{A}$.

4 . b) En utilisant l'algorithme de déterminisation vu en cours, construire un automate déterministe $\mathcal{B}$ qui reconnaît le même langage que $\mathcal{A}$.

Exercice II. Compilation À rendre sur une copie séparée

On souhaite ajouter à notre calculette le support des pointeurs vers les entiers. Pour cela, nous introduisons deux opérateurs:

- `&a` renvoie la position dans la pile de la variable `a`;
- `*x` renvoie le contenu de la pile en position `x`;

Pour déclarer un pointeur, on utilisera le mot clef `ptr`. Par exemple:

```
ptr p
```

On utilise l'opcode **LOAD** qui charge la valeur de la pile dont l'adresse est indiquée sur le haut de la pile.

Code	Pile	*sp*	*pc*
LOAD	P[sp-1] := P[P[sp-1]]	sp	pc+1

Question 5. Quelles valeurs affiche le programme suivant lorsque l'on l'exécute? (On demande les deux entiers affichés, pas le MVàP)

```
int x = 40
ptr p
p = &x
write(p)
write(*p)
```

On se donne le code suivant:

```
int x
int y = 42
int z = 3
ptr p

p = &y
for (x = 0 ; x < 2 ; x++) {
write(p+x)
write(*(p+x))
}
```

qui se traduit en MVàP par

```
LABEL Main
  PUSHI 0
  PUSHI 42
  PUSHI 3
  PUSHI 0

  PUSHI 1
  STOREG 3

  PUSHI 0
  STOREG 0
LABEL Label1
  PUSHG 0
  PUSHI 2
  INF
  JUMPF Label2
  PUSHG 3
  PUSHG 0
  ADD
  WRITE
  POP
  PUSHG 3
  PUSHG 0
  ADD
  LOAD
  WRITE
  POP
  PUSHG 0
  PUSHI 1
  ADD
  STOREG 0
  JUMP Label1
LABEL Label2
HALT
```

et le résultat de son assemblage

```
 Adr |  Instruction
-----+---------------
   0 | PUSHI       0
   2 | PUSHI      42
   4 | PUSHI       3
   6 | PUSHI       0
   8 | PUSHI       1
  10 | STOREG      3
  12 | PUSHI       0
  14 | STOREG      0
  16 | PUSHG       0
  18 | PUSHI       2
  20 | INF
  21 | JUMPF      47
  23 | PUSHG       3
  25 | PUSHG       0
  27 | ADD
  28 | WRITE
  29 | POP
  30 | PUSHG       3
  32 | PUSHG       0
  34 | ADD
  35 | LOAD
  36 | WRITE
  37 | POP
  38 | PUSHG       0
  40 | PUSHI       1
  42 | ADD
  43 | STOREG      0
  45 | JUMP       16
  47 | HALT
```

Question 6. Compléter la trace d'exécution suivante.

```
  pc |               |    fp   pile
 ================================================
   0 | PUSHI       0 |     0 [ ] 0
   2 | PUSHI      42 |     0 [ 0 ] 1
   4 | PUSHI       3 |     0 [ 0 42 ] 2
   6 | PUSHI       0 |     0 [ 0 42 3 ] 3
   8 | PUSHI       1 |     0 [ 0 42 3 0 ] 4
  10 | STOREG      3 |     0 [ 0 42 3 0 1 ] 5
  12 | PUSHI       0 |     0 [ 0 42 3 1 ] 4
  14 | STOREG      0 |     0 [ 0 42 3 1 0 ] 5
  16 | PUSHG       0 |     0 [ 0 42 3 1 ] 4
  18 | PUSHI       2 |     0 [ 0 42 3 1 0 ] 5
  20 | INF           |     0 [ 0 42 3 1 0 2 ] 6
  21 | JUMPF      47 |     0 [ 0 42 3 1 1 ] 5
  23 | PUSHG       3 |     0 [ 0 42 3 1 ] 4
  25 | PUSHG       0 |     0 [ 0 42 3 1 1 ] 5
  27 | ADD           |     0 [ 0 42 3 1 1 0 ] 6
  28 | WRITE         |     0 [ 0 42 3 1 1 ] 5
     1
  29 | POP           |     0 [ 0 42 3 1 1 ] 5
  30 | PUSHG       3 |     0 [ 0 42 3 1 ] 4
  32 | PUSHG       0 |     0 [ 0 42 3 1 1 ] 5
  34 | ADD           |     0 [ 0 42 3 1 1 0 ] 6
  35 | LOAD          |     0 [ 0 42 3 1 1 ] 5
  36 | WRITE         |     0 [ 0 42 3 1 42 ] 5
    42
  37 | POP           |     0 [ 0 42 3 1 42 ] 5
  38 | PUSHG       0 |     0 [ 0 42 3 1 ] 4
  40 | PUSHI       1 |     0 [ 0 42 3 1 0 ] 5
  42 | ADD           |     0 [ 0 42 3 1 0 1 ] 6
  43 | STOREG      0 |     0 [ 0 42 3 1 1 ] 5
  45 | JUMP       16 |     0 [ 1 42 3 1 ] 4


  <...>


  42 | ADD           |     0 [ 1 42 3 1 1 1 ] 6
  43 | STOREG      0 |     0 [ 1 42 3 1 2 ] 5
  45 | JUMP       16 |     0 [ 2 42 3 1 ] 4
  16 | PUSHG       0 |     0 [ 2 42 3 1 ] 4
  18 | PUSHI       2 |     0 [ 2 42 3 1 2 ] 5
  20 | INF           |     0 [ 2 42 3 1 2 2 ] 6
  21 | JUMPF      47 |     0 [ 2 42 3 1 0 ] 5
  47 | HALT          |     0 [ 2 42 3 1 ] 4
```

Question 7. Écrire le code MVàP correspondant à la fonction suivante :

```
var x : int = 0
ptr p

fun calc ( y : int) : int {
  return *(p+1) + y
}

p = &x
write(calc(3))
```

On donne la grammaire suivante permettant de prendre en compte la déclaration de pointeurs:

```
declaration returns [ String code ]
    <...>
    | 'ptr' IDENTIFIANT
      {  // À compléter }
    <...>
    ;
```

Question 8. Compléter les actions de la grammaire afin de générer le code MVàP correct.

On se donne également la grammaire de l'utilisation des pointeurs :

```
expr returns [ String code ]
    <...>
    | '&' IDENTIFIANT
      {  // À compléter }
    | '*' EXPR
      {  // À compléter }
    <...>
    ;
```

Question 9. On suppose pour cette question que le pointeur concerne uniquement des variables globales. Compléter les actions de la grammaire afin de générer le code MVàP correct.

On autorise désormais le pointeur à utiliser (en plus) des variables locales. Dans ce cas, le pointeur contient toujours la position (forcément positive) de la variable dans la pile.

Question 10. Indiquer quelles sont les actions de la grammaire à modifier.

On ajoute les opcodes **PUSHFP** (**PUSHSP**) qui mettent respectivement le registre *fp* (*sp*) sur le haut de la pile.

Code	Pile	*sp*	*pc*
PUSHFP	P[sp] := fp	sp + 1	pc+1
PUSHSP	P[sp] := sp	sp + 1	pc+1

Question 11. Compléter les actions de la grammaire afin de générer le code MVàP correct.

Exercice III. Analyse LL — À rendre avec l'exercice I

Soit la grammaire $\mathcal{G}$ d'axiome S avec six symboles terminaux { [,] , c , id , int , float }, et définie par :

$$\begin{cases} \texttt{S} \rightarrow \texttt{A id I} \\ \texttt{I} \rightarrow \texttt{[T] I} \mid \varepsilon \\ \texttt{T} \rightarrow \texttt{c} \mid \varepsilon \\ \texttt{A} \rightarrow \texttt{int} \mid \texttt{float} \end{cases}$$

Question 12. Donner un arbre d'analyse pour le mot suivant :

```
int id [ ] [ ]
```

On dispose des tables Effacable, Premier et Suivant de la grammaire $\mathcal{G}$:

Symbole	Effacable	Premier	Suivant
S	Non	int , float	$
I	Oui	[	$
T	Oui	d	]
A	Non	int , float	id

Question 13.

13.a) Indiquer de façon précise comment l'ensemble Premier(S) est déterminé.

13.b) Expliquer de même comment l'ensemble Suivant(I) est obtenu.

Question 14. Construire la table d'analyse LL(1) de la grammaire $\mathcal{G}$. Distinguer les règles qui mettent en jeu les ensembles Premier des règles qui mettent en jeu les ensembles Suivant.

Question 15.

15.a) À l'aide de cette table d'analyse LL(1), dérouler l'analyse sur l'entrée

```
int id [ ]
```

15.b) De même, dérouler l'analyse LL(1) sur l'entrée

```
int id [ c c ]
```