Exercice 1 : Grammaires (15 pts)

1.1. Écrire une grammaire capable de reconnaître une expression composée de parenthèses et d'identificateurs **id** telle que les parenthèses sont valides (une parenthèse ouverte est toujours fermée et inversement). Il ne peut pas y avoir deux identificateurs à la suite.

Les expressions suivantes sont acceptables :

- id
- ( )
- id (id ( ) id (id))

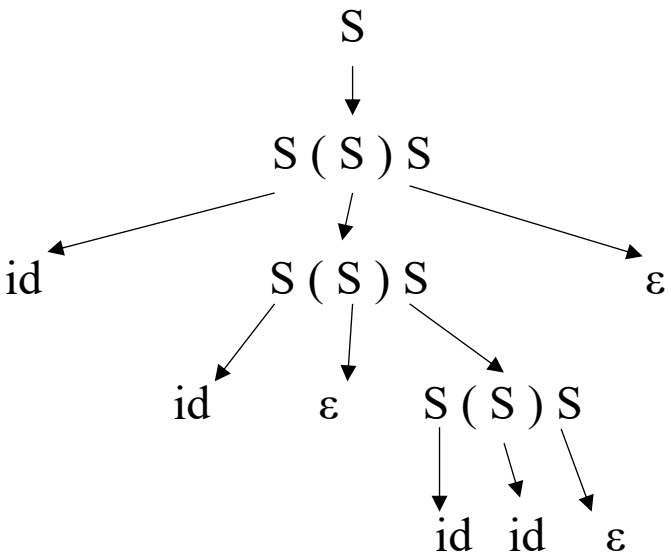
Les expressions suivantes sont invalides :

- (id id) **deux identificateurs à la suite**
-) **parenthèse non ouverte**
- (id ( ) id (id) **parenthèse non fermée**

Plusieurs solutions possibles, la plus simple étant : S => S (S) S | id | ε

1.2. À l'aide de votre grammaire, construire l'arbre syntaxique de l'expression :

id (id ( ) id (id))



Exercice 2 : Parseur descendant (35 pts)

Soit la grammaire suivante représentant un petit langage de programmation (fictif) :

(1) P → **begin** L **end**
(2) L → L I
(3) L → I
(4) I → S **id** ;
(5) I → **id** = E ;
(6) I → **print** E ;
(7) I → **while** (C) { L }
(8) S → **int**
(9) S → **float**
(10) E → E + T
(11) E → E - T
(12) E → T
(13) T → T * F
(14) T → F
(15) S → **id**
(16) S → **num**
(17) S → (E)
(18) C → E **binary_op** E

Les terminaux sont ici **begin, end, id, print, while, int, float, num, binary_op, (,), {, }** et **=**.

2.1. Éliminer la récursivité à gauche de cette grammaire, s'il y a lieu. Ne réécrivez pas toute la grammaire, seulement les règles modifiées.

2 et 3 deviennent :

L => I L'

L' => I L' | ε

10, 11 et 12 deviennent :

E => T E'

E' => + T E' | - T E' | ε

13 et 14 deviennent :

T => F T’

T’ => * F T’ | ε

2.2. Calculez les ensembles FIRST et FOLLOW des non-terminaux de la grammaire obtenue au 2.1. Pour vous aider, certains ensembles sont fournis.

FIRST(P) = { **begin** }

FIRST(L) = FIRST(I) = { **int, float, id, print, while** }

FOLLOW(P) = {**$}**

FOLLOW(L) = {**end, }** }

Non-terminal	FIRST	FOLLOW
I	(donné)	int, float, id, print, while, end, }
S	int, float	id
E	id, num, (	; , binary_op,)
T	id, num, (	+, -, ; , binary_op,)
F	id, num, (	*, +, -, ; , binary_op,)
C	id, num, (	)
E’	+, -, ε	; , binary_op,)
T’	*, ε	+, -, ; , binary_op,)
L’	id, print, while, int, float, ε	end, }

2.3. Nous allons implémenter une partie d'un parseur LL permettant de reconnaître ce langage, comme vu en cours. La variable *lookahead* contient le terminal courant dans l'entrée. On vous fournit la fonction *match* :

```
void match (terminal t) {
     if (lookahead == t) {
          lookahead = nextTerminal ();
     } else {
          error ();
     }
}
```

Votre code doit s'assurer de renvoyer un message d'erreur approprié lorsque nécessaire.

Vous pouvez écrire dans le langage usuel de votre choix ou en pseudo-code.

Complétez les fonctions suivantes pour obtenir un parseur LL, comme vu en cours :

```
void I () {

     switch lookahead {
          case id
               match(id) ; match(=) ; E() ; match(;) ;
               break ;
          case print
               match(print) ; E() ; match(;) ; break ;
          case while
               match(while) ; match( ( ) ; C() ;
               match( ) ) ; match( { ) ; L() ; match( } ) ;
               break ;
          case int || float
               S() ; match(id) ; match(;) ; break
          default
               error('Symbole non attendu reçu') ;
     }
}
```

```
void E () {
     if (lookahead == 'id' || lookahead == 'num' ||
     lookahead == ')') {
          T() ; E_prime() ;
     else {
          error('Symbole non attendu reçu') ;
     }
}
```

Exercice 3 : Parseur ascendant (25 pts)

Nous allons travailler sur la grammaire :

$$
\begin{aligned}
&(1)\ \mathrm{S'} \rightarrow \mathrm{S} \\
&(2)\ \mathrm{S} \rightarrow \mathrm{A\ B} \\
&(3)\ \mathrm{A} \rightarrow \mathbf{a}\mathrm{A} \\
&(4)\ \mathrm{A} \rightarrow \mathbf{a} \\
&(5)\ \mathrm{B} \rightarrow \mathbf{b}\mathrm{B} \\
&(6)\ \mathrm{B} \rightarrow \mathbf{b}
\end{aligned}
$$

3.1. Calculer les ensembles :

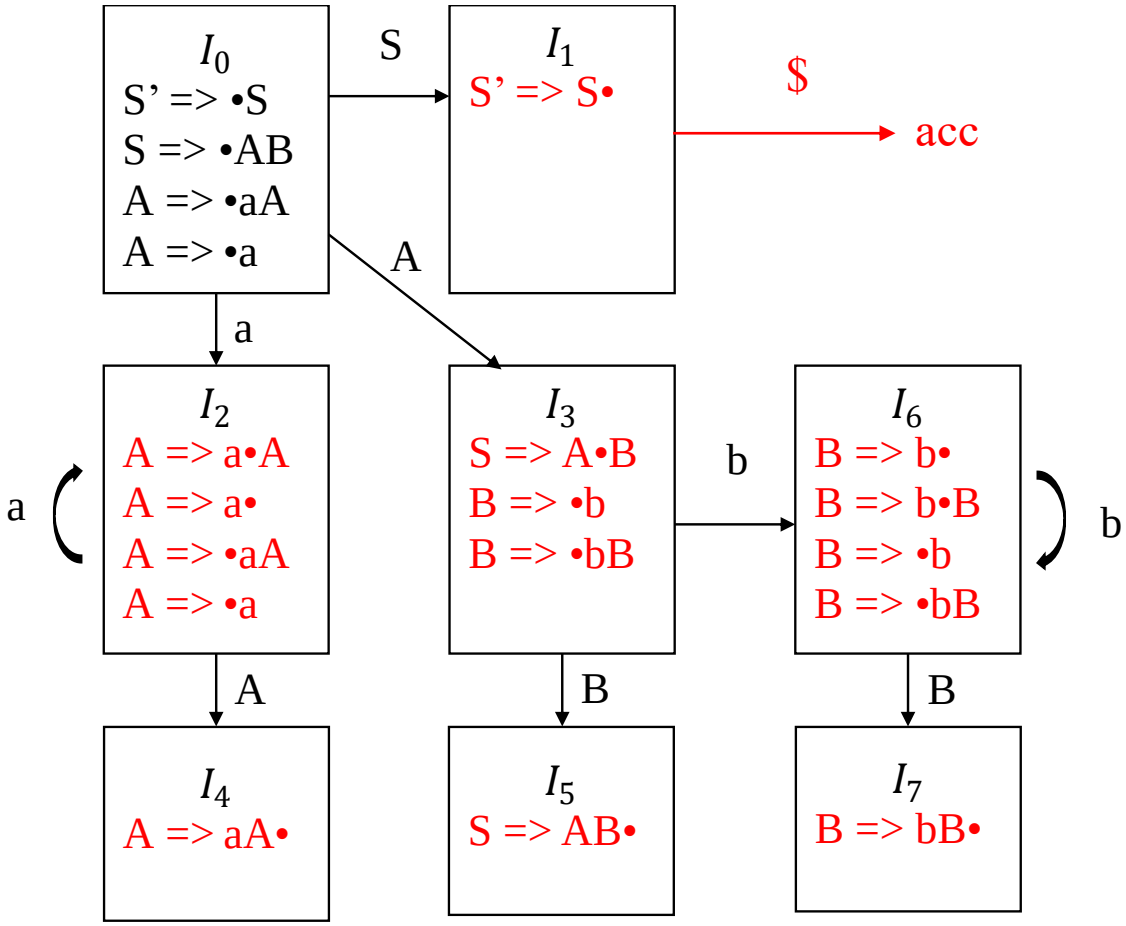
FOLLOW(S') = {$}

FOLLOW(S) ={$}

FOLLOW(A) ={b}

FOLLOW(B) = {$}

3.2. Construire l'automate LR(0) associé à cette grammaire. I_0 est fourni, ainsi que la structure de l'automate.



3.3. Remplir la table SLR de cette grammaire en utilisant les conventions du cours (sX, rY, acc).

État	Action			Goto		
	a	b	$	S	A	B
0	s2			1	3	
1			acc			
2	s2	r4			4	
3		s6				5
4		r3				
5			r2			
6		s6	r6			7
7			r5			

3.4. Compléter la table d'actions pour la chaine **ab** :

Étape	Pile	Entrée	Action
0	$0	ab$	s2
1	$0 2	b$	r4
2	$0 3	b$	s6
3	$0 3 6	$	r6
4	$0 3 5	$	r2
5	$0 1	$	acc

Exercice 4 : Traduction dirigée par la syntaxe (15 pts)

Soit la grammaire suivante, représentant une série d'initialisation de variables suivie d'une série de calculs, composés uniquement d'additions, sur ces variables :

(1) S => A I
(2) A => **int id ;** A
(3) A => ε
(4) I => **id =** E **;** I
(5) I => ε
(6) E => E + E
(7) E => **id**
(8) E => **num**

4.1. On dispose de deux fonctions : *add(*id*)* qui ajoute *id* à la liste des variables qui ont été initialisées, et *check*(id) qui vérifie que *id* a bien été initialisé, et renvoie un message d'erreur si non.

Écrire un schéma de traduction dirigé par la syntaxe (SDT) pour cette grammaire, appelant les fonctions *add* et *check* afin de vérifier que toute variable utilisée dans une instruction a bien été initialisée. Il faut que le message d'erreur apparaisse pour la première variable non valide rencontrée : dans l'expression

```
int X ;

int Y ;

Y = Z + X ;

T = Y + 2 ;
```

il faut que le message d'erreur apparaisse d'abord pour Z, et ensuite pour T.

(1) S => A I
(2) A => **int id** {add(id)} **;** A
(3) A => ε
(4) I => **id** {check(id)} **=** E **;** I
(5) I => ε
(6) E => E + E
(7) E => **id** {check(id)}

(8) E => **num**

4.2. On dispose cette fois de deux fonctions : *setVal*(id, val) qui indique que la variable *id* a la valeur *val*, et *getVal*(id) qui renvoie la valeur de *id* si elle existe, une erreur sinon.

Écrire une définition dirigée par la syntaxe (SDD) permettant de mettre à jour la valeur des variables au fur et à mesure.

Règle de production	Règle sémantique
(4) I => id = E ; I	setVal(id, E.val)
(6) E => E_1 + E_2	E.val = E_1.val + E_2.val
(7) E => id	E.val = getVal(id)
(8) E => num	E.val = num.lexval

4.3. Votre grammaire est-elle S-attribuée ? L-attribuée ? Justifier les deux réponses.

La grammaire est S-attribuée car le seul attribut, E.val, est synthétisé.

La grammaire est également L-attribuée car S-attribuée => L-attribuée.

Exercice 5 : JavaCC (10 pts)

Voici un extrait de code javaCC :

```
PARSER_BEGIN(Function)

public class Function {
    public static void main(String[] args) {
        try {
            new Function(new
java.io.StringReader(args[0])).S();
            System.out.println("Okay");
        } catch (Throwable e) {
            System.out.println("Caught something");
        }
    }
}

PARSER_END(Function)

SKIP:  { " " | "\t" | "\n" | "\r"                    }
TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }

void S(): {} { E() <EOF>           }
void E(): {} { T() ("+" T())*      }
void T(): {} { F() ("*" F())*      }
void F(): {} { <NUM> | "(" E() ")" }
```

5.1. Décrire en quelques mots les différents blocs de ce code :

de PARSER_BEGIN à PARSER_END :

Corps du programme. Appelle la première fonction S( ) sur l'entrée, indiquant que S est le non-terminal de départ. Si une erreur est retournée quelque part dans la grammaire, elle est attrapée ici et l'on peut afficher un message.

la ligne SKIP :

Indique les caractères qui sont ignorés par le parseur.

la ligne TOKEN :

Contient les terminaux de la grammaire, sous forme d'expression régulière : (,), +, *, et le terminal NUM, constitué de chiffres (au moins un) entre 0 et 9.

les quatre dernières lignes :

Représente les règles de production de la grammaire.

5.2. Quelle est la grammaire reconnue ?

La grammaire des opérations arithmétiques + et * sur des nombres entiers (positifs), avec priorité des opérateurs incluant des parenthèses.

Bon courage ! ☺