TP3 : L'ANALYSE SYNTAXIQUE POUR LE LANGAGE A.

1. OBJECTIF

L'objectif de ce TP est de programmer un analyseur syntaxique pour le langage A, destiné à la description d'automates finis déterministes. Dans ce projet, l'analyseur syntaxique reçoit le flux de tokens envoyé par la phase d'analyse lexicale précédente, sous forme d'un seul mot. Il analyse ensuite ce mot afin de vérifier s'il appartient à la grammaire qui engendre le langage A. Notre tâche dans ce TP est donc d'écrire les règles grammaticales correspondant à tout programme correct du langage A.

2. LA SYNTAXE GENERALE DU LANGAGE A DE DESCRIPTION D'AUTOMATES FINIS DETERMINISTE.

La syntaxe proposée pour le langage A est présentée ci-dessous :

1. **Déclaration de l'automate :**
   ```
   automate <NomAutomate> { … }
   ```
2. **Déclaration de l'alphabet de l'automate :**
   ```
   alphabet = { <symbole1> [, <symbole2> ... ] };
   ```
3. **Les états de l'automate :**
   ```
   etats = { <etat1> [, <etat2> ... ] };
   ```
4. **Déclaration de l'état initial :**
   ```
   initial = <etat_initial>;
   ```
5. **Déclaration des états finaux :**
   ```
   finaux = { <etat_final1> [, <etat_final2> ... ] };
   ```
6. **Les transitions :**
   ```
   transitions = {
       <etat_source> : <symbole> -> <etat_destination>;
       [ ... autres transitions ... ]
   };
   ```
7. **Vérification d'un mot :**

```
verifier <NomAutomate> "<mot>";
```

Voici la syntaxe générale globale d'un programme en langage A :

```
automate <NomAutomate> {
    alphabet = { <symbole1> [, <symbole2> ... ] };
    etats = { <etat1> [, <etat2> ... ] };
    initial = <etat_initial>;
    finaux = { <etat_final1> [, <etat_final2> ... ] };
    transitions = {
        <etat_source> : <symbole> -> <etat_destination>;
        [ <etat_source> : <symbole> -> <etat_destination>; ... ]
    };
}

verifier <NomAutomate> "<mot>";
```

3. CE QU'IL FAUT FAIRE

1. Sauvegarder une copie de votre projet actuel dans votre dossier de travail.
2. Ouvrir le dossier du projet où vous avez déjà réalisé la phase d'analyse lexicale.
3. Créer un nouveau fichier parser.y.
4. Définir tous les tokens utilisés dans la phase lexicale avec l'instruction %token, par exemple :

   ```
   %token ID ACC_OUVR ACC_FER POINT_VIRGULE ...
   ```

5. Définir la grammaire du langage A dans parser.y sous la forme :

   ```
   program:
       program automate_decl
       | program verification
   ;
   ```

   … puis continuer avec toutes les règles de votre langage (alphabet, états, transitions…).

6. Gérer les erreurs syntaxiques avec la fonction yyerror et afficher la ligne + token fautif :

```
extern int yylineno;
extern char *yytext;
void yyerror(const char *msg) {
    printf("Ligne %d, près de '%s' : %s\n", yylineno, yytext,
msg);
}
```

7. Ajouter la fonction main() dans parser.y pour analyser un fichier au lieu d'utiliser la console. Utiliser yyparse() au lieu de yylex() :

```
int main(int argc, char **argv) {
    FILE *file = fopen("code.a","r");
    if (!file) {
        perror("Erreur ouverture fichier");
        return 1;
    }
    yyin = file;
    yyparse();
    fclose(file);
    return 0;
}
```

8. Enregistrer parser.y puis générer les fichiers Bison :

```
bison -d parser.y
```

Cela crée parser.tab.h et parser.tab.c.

9. Modifier l'analyseur lexical (lexer.l) :
    a. Supprimer la fonction main() si elle existe.
    b. Ajouter au début du fichier : #include "parser.tab.h".
    c. Modifier les actions des expressions régulières pour retourner les tokens au lieu de les afficher avec printf.
10. Enregistrer sous lexer.l, puis générer le code Flex :

```
flex lexer.l
```

11. Compiler les fichiers Flex et Bison ensemble :

```
gcc -o ensa parser.tab.c lex.yy.c -lfl
```