Licence 2 Sciences et Technologie **UGA**
Parcours Informatique **UE INF404**

Examen du 24 mai 2023

Durée : 2h - Une feuille A4 recto-verso autorisée - Calculatrices interdites

Les programmes demandés peuvent être écrits en C et/ou en notation algorithmique.

Introduction

On s'intéresse à un langage $\mathcal{L}$ de description de données pour GPS. Ces données sont constituées :

- d'une séquence de **points**, où chaque point est défini par un *nom* auquel est associé un couple de *coordonnées* dans un repère othonormé ;
- d'une séquence de **traces**, où chaque trace est définie par un *nom* auquel est associée une description de la trace sous forme de concaténations et répétitions de **segments**, et où chaque segment est soit une séquences de points soit une trace déjà définie précédemment.

On donne ci-dessous un exemple de programme écrit dans ce langage. Les commentaires (précédés du caractère `#`) ne font pas partie du langage et sont fournis pour aider à comprendre l'exemple.

```
Points   # séquence de 4 points
  P1 = (12,4) ; # ce point a pour nom P1 et pour coordonnées (12,4)
  P12 = (0,9) ;
  P3 = (18,25) ;
  P5 = (42,42)
Traces   # séquence de 3 traces
  T1 = [P1, P5, P3] ;  # séquence des 3 points P1, P5 et P3
  T8 = T1.[P12, P1] ;  # concaténation de T1 avec le segment [P12, P1]
  T9 = T1 5 . T8       # répétition 5 fois de T1 et concaténation avec T8
```

Dans cet examen nous nous intéressons à l'analyse lexicale de ce langage (Question 1), puis à son analyse syntaxique (Question 2). Nous étendrons ensuite l'analyse syntaxique afin de produire des structures intermédiaires (Question 3 et 4). Nous terminerons alors (Question 5) par le programme principal et une extension envisageable.

Question 1 : analyse lexicale [$\sim$ 3 points]

Les lexèmes du langage sont les suivants :

- `POINTS`, représente le mot-clé "`Points`" ;
- `TRACES`, représente le mot-clé "`Traces`" ;
- `NPOINT`, représente un *nom de point*, constitué de la lettre '`P`' suivie d'un entier, comme `P12` ;
- `NTRACE`, représente un *nom de trace*, constitué de la lettre '`T`' suivie d'un entier, comme `T2` ;
- `EGAL`, représente le caractère '`=`' ;
- `PARO`, représente le caractère '`(`' ;
- `PARF`, représente le caractère '`)`' ;
- `CROCHO`, représente le caractère '`[`' ;
- `CROCHF`, représente le caractère '`]`' ;

— `VIRG`, représente le caractère '`,`';
— `PVIRG`, représente le caractère '`;`';
— `DOT`, représente le caractère '`.`';
— `ENTIER`, représente un entier positif (une suite non vide de chiffres).

On suppose que les caractères "espace" et "fin ligne" sont des séparateurs.

1. Donnez un exemple de lexème incorrect.
2. Dessinez un automate de reconnaissance des lexèmes du langage. Cet automate lit en entrée une séquence de caractères et il atteint un état final lorsqu'un lexème a été reconnu.

Question 2 : syntaxe du langage [$\sim$ 2 points]

La grammaire complète du langage est la suivante :

$$
\begin{array}{rcl}
Pgm & \rightarrow & \texttt{POINTS}\ SeqPts\ \texttt{TRACES}\ SeqTraces\ \texttt{FSEQ} \\
SeqPts & \rightarrow & Point\ SuiteSeqPts \\
SuiteSeqPts & \rightarrow & \texttt{PVIRG}\ \ SeqPts \\
SuiteSeqPts & \rightarrow & \varepsilon \\
Point & \rightarrow & \texttt{NPOINT EGAL PARO ENTIER VIRG ENTIER PARF} \\
SeqTraces & \rightarrow & Trace\ SuiteSeqTraces \\
SuiteSeqTraces & \rightarrow & \texttt{PVIRG}\ SeqTraces \\
SuiteSeqTraces & \rightarrow & \varepsilon \\
Trace & \rightarrow & \texttt{NTRACE EGAL}\ SeqSegment \\
SeqSegment & \rightarrow & Segment\ SuiteSeqSegment \\
SuiteSeqSegment & \rightarrow & \texttt{DOT}\ Segment\ SuiteSeqSegment \\
SuiteSeqSegment & \rightarrow & \varepsilon \\
Segment & \rightarrow & SegElementaire\ Repetition \\
Repetition & \rightarrow & \texttt{ENTIER} \\
Repetition & \rightarrow & \varepsilon \\
SegElementaire & \rightarrow & \texttt{NTRACE} \\
SegElementaire & \rightarrow & \texttt{CROCHO NPOINT}\ SuiteSeqChaine \\
SuiteSeqChaine & \rightarrow & \texttt{CROCHF} \\
SuiteSeqChaine & \rightarrow & \texttt{VIRG NPOINT}\ \ SuiteSeqChaine
\end{array}
$$

1. Le langage $\mathcal{L}$ est-il un langage *régulier*? Justifiez (informellement) votre réponse.
2. Donnez un exemple de programme syntaxiquement incorrect.

Question 3 : construction d'une table de points [$\sim$ 5 points]

Il est conseillé de lire l'ensemble de cette question avant de répondre.

Cette partie consiste à anlayser la partie "séquence de points" du langage afin de construire une *table des points* permettant d'associer à chaque *nom de point* le couple de coordonnées qui lui correspond.

On donne ci-dessous la table des points attendue pour l'exemple de programme donné en introduction.

P1	(12,4)
P12	(0,9)
P3	(18,25)
P5	(42,42)

1. Donnez le contenu du fichier `table_points.h` contenant la **déclaration** de votre table des points ainsi que les **en-têtes** de fonctions permettant :
   — d'insérer un point dans cette table (avec son nom et ses coordonnées)
   — de consulter si un point dont le nom est spécifié en paramètre est présent dans la table
   — de consulter les coordonnées d'un point (présent dans la table) dont le nom est spécifié en paramètre.
2. En utilisant les primitives du module `analyse_lexicale.h`, fourni en Annexe A, écrivez le corps de la procédure `consTabPoints` (spécifiée ci-dessous), qui effectue une analyse syntaxique d'une séquence de point et construit – s'il n'y a pas d'erreurs – la table de points correspondante. Vous pouvez utiliser des procédures auxilliaire (comme `Rec_SeqPts`, etc.). **Attention**, il n'est **pas nécessaire** dans cette question d'analyser la partie du fichier correspondant à la *séquence de traces*.

```
void consTabPoints (char *nomfichier) ;
// lit une séquence de points dans le fichier nomfichier et construit la table de points associée
// - si cette séquence est syntaxiquement correcte
// - et elle ne contient pas deux points de même nom
// sinon on appelle la fonction Erreur().
```

Question 4 : construction d'une table de traces [∼ 6 points]

Il est conseillé de lire l'ensemble de cette question avant de répondre.

Cette partie consiste à analyser la partie "séquence de traces" du langage afin de construire une *table des traces* permettant d'associer à chaque *nom de trace* sa **longueur**, définie comme le *le nombre de points que l'on doit traverser pour parcourir cette trace.*

On donne ci-dessous la table des traces attendue pour l'exemple de programme donné en introduction.

T1	3
T8	5
T9	20

— la longueur de T1 est **3** (3 points)
— la longueur de T8 est **5** (la longueur de T1 plus 2 points)
— la longueur de T9 est **20** (la longueur de T1 multipliée par 5 plus la longueur de T8)

1. Donnez le contenu du fichier `table_trace.h` contenant la **déclaration** de votre table des traces ainsi que les **en-têtes** de fonctions permettant :
   — d'insérer une trace dans cette table (avec son nom et sa longeur)
   — de consulter si une trace dont le nom est spécifié en paramètre est présente dans la table
   — de consulter la longueur d'une trace (présente dans la table) dont le nom est spécifié en paramètre.
2. En utilisant les primitives du module `analyse_lexicale.h`, fourni en Annexe A, écrivez le corps de la procédure `consTabTraces` (spécifiée ci-dessous), qui effectue une analyse syntaxique d'une séquence de traces et construit – s'il n'y a pas d'erreurs – la table de points correspondante. Vous pouvez utiliser des procédures auxilliaire (comme `Rec_SeqTraces`, etc.). **Attention**, il n'est **pas nécessaire** d'analyser la partie du fichier correspondant à la *séquence de points*, par contre il faudra consulter la table de points construite à la Question 3.

```
void consTabTraces (char *nomfichier) ;
// lit une séquence de traces dans le fichier nomfichier et construit la table de traces associée
// - si cette séquence est syntaxiquement correcte
// - et si elle ne contient pas deux traces de même nom
// - et si toute trace est définie avant d'etre utilisée
// - et si toute trace ne contient que des points présents dans la table des points
// sinon on appelle la fonction Erreur().
```

Indications :
Pour calculer la longueur des traces composées de concaténation ou répétitions de segments vous pouvez construire un *arbre abstrait*, comme vu en projet dans le cas d'une séquence d'affectations. Ce n'est toutefois pas indispensable, vous pouvez aussi calculer "directement" ces longueurs lors de l'analyse syntaxique (quitte à utiliser des paramètres supplémentaires).

Question 5 : programme principal et extension [$\sim$ 4 points]

1. Ecrivez un programme principal permettant
   (a) d'ouvrir le fichier dont le nom est fourni en argument de la ligne de commande ;
   (b) de construire la table des points ;
   (c) de construire la table des traces.
2. On considére maintenant une définition (plus réaliste !) de la longueur d'une trace comme la distance euclidienne qu'il faut parcourir pour relier l'ensemble des points de cette trace (en prenant en compte les concaténations et les répétitions de segments). On rapelle que la distance euclidienne entre deux points de coordonnées (x_1, y_1) et (x_2, y_2) est définie par la formule :

   $$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

   Indiquez comment modifier votre réponse à la Question 4 pour prendre en compte cette nouvelle définition.

Annexe A : `analyse_lexicale.h`

```
typedef enum  {
POINTS, TRACES, NPOINT, NTRACE, PARO, PARF, CROCHO, CROCHF, VIRG, PVIRG, DOT, ENTIER, EGAL
} Nature_Lexeme ;

typedef struct {
   Nature_Lexeme nature; // nature du lexeme
   char chaine[NBCAR] ;   // chaine de caractere
   int valeur;          // valeur pour un lexeme ENTIER
} Lexeme ;

void demarrer(char *nom_fichier);// initialise l'analyse lexicale
void avancer();// lit le lexeme suivant
Lexeme lexeme_courant();    // pourra etre abrege en "LC"
int fin_de_sequence();   // vrai ssi la fin de sequence est atteinte
```